TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair

Deheng Yang
yangdeheng13@nudt.edu.cn
National University of Defense Technology
China

Xiaoguang Mao*
xgmao@nudt.edu.cn
National University of Defense Technology
China

Liqian Chen
lqchen@@nudt.edu.cn
National University of Defense Technology
China

Xuezheng Xu
xuezhengxu@126.com
Academy of Military Sciences
China

Yan Lei
yanlei@cqu.edu.cn
Chongqing University
China

David Lo
davidlo@smu.edu.sg
Singapore Management University
Singapore

Jiayu He
hejy47@nudt.edu.cn
National University of Defense Technology
China

ABSTRACT

Automated program repair (APR) holds the promise of aiding manual debugging activities. Over a decade of evolution, a broad range of APR techniques have been proposed and evaluated on a set of real-world bug datasets. However, while more and more bugs have been correctly fixed, we observe that the growth of newly fixed bugs by APR techniques has hit a bottleneck in recent years. In this work, we explore the possibility of addressing complicated bugs by proposing *TransplantFix*, a novel APR technique that leverages graph differencing-based transplantation from the donor method. The key novelty of *TransplantFix* lies in three aspects: 1) we propose to use a graph-based differencing algorithm to distill semantic fix actions from the donor method; 2) we devise an inheritance-hierarchy-aware code search approach to identify donor methods with similar functionality; 3) we present a namespace transfer approach to effectively adapt donor code.

We investigate the unique contributions of *TransplantFix* by conducting an extensive comparison that covers a total of 42 APR techniques and evaluating *TransplantFix* on 839 real-world bugs from Defects4J v1.2 and v2.0. *TransplantFix* presents superior results in three aspects. First, it has achieved the best performance as compared to the state-of-the-art APR techniques proposed in the last three years, in terms of the number of newly fixed bugs, reaching a 60%-300% improvement. Furthermore, not relying on any fix actions crafted manually or learned from big data, it reaches the best generalizability among all APR techniques evaluated on Defects4J v1.2 and v2.0. In addition, it shows the potential to synthesize complicated patches consisting of at most eight-line insertions at a hunk. *TransplantFix* presents fresh insights and a promising avenue for follow-up research towards addressing more complicated bugs.

*Xiaoguang Mao is the corresponding author.

ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556893

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Automated program repair, graph differencing, code transplantation

ACM Reference Format:
Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556893

1 INTRODUCTION

Automated program repair (APR for short), which aims to produce bug fixes without human intervention [49], is regarded as a promising technique to alleviate the heavy burden of manual debugging activities. In the APR community, fix ingredients are a commonly used conception, which initially denoted the code fragment needed to synthesize a patch (aka donor code) [48, 68]. In recent years, fix actions, which encode instructions to modify source code at the abstract level, have also been included as a type of fix ingredients [22, 76, 80]. In the last decade, a broad range of APR techniques have been proposed to better leverage fix actions and donor code

[5, 22, 23, 29, 37, 67, 78, 82]. These techniques have achieved promising results in real-world bug datasets [24, 32] or even applied to industrial pipelines [7, 45, 50, 64].

While an increasing number of real-world bugs have been fixed by existing APR techniques, we observe that the growth of newly fixed bugs by APR techniques is hitting a bottleneck via a literature review. Figure 1 presents the trend of newly fixed bugs on a commonly evaluated dataset (i.e., Defects4J v1.2 [24][1]). As shown in the figure, the number of newly fixed bugs has experienced rapid growth from 2016 to 2019. However, since 2020, the growth has become much slower (e.g., there are only two new bugs are fixed in 2022). As reported in prior studies [25, 39], the bugs that are still not fixed by APR techniques tend to be "complicated" in several perspectives, e.g., involving multiple inserted lines (i.e., donor code) [25] or multiple statements impacted by change actions (i.e., fix actions) [39].

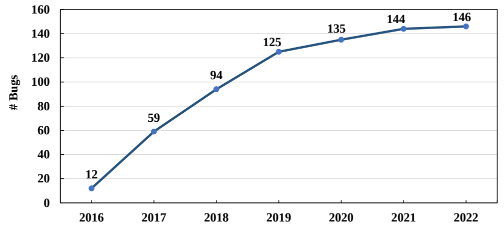

Figure 1: The total number of bugs that have been correctly fixed by APR techniques on 395 bugs in Defects4J v1.2. Repair data of 36 APR techniques proposed between 2016 and 2022 and evaluated on Defects4J are considered.

We summarize three challenges that hinder APR techniques to repair more new bugs from the perspective of fix ingredients: 1) localization of fix ingredients; 2) extraction of fix actions; 3) adaption of donor code. The first challenge is ***where we can find complicated fix ingredients for patch synthesis***. As for donor code, existing APR techniques have explored a series of settings, including searching donor code within the buggy file [37, 67], buggy project [22], or even external projects [5, 38, 71]. In terms of fix actions, existing APR techniques generally assume that fix actions for fixing bugs reside in real-world human-written bug fixes of external projects, which can be mined manually [27] or learned automatically [10, 33, 82]. While fix ingredients are found to fix a number of single-line or single-chunk bugs [39], where the fix ingredients could be found for larger bug fixes still remains an unanswered question.

The second challenge is ***how to distill complicated fix actions***. Existing APR techniques distill fix actions via multiple approaches (e.g., manual summarization [14, 20, 27, 56, 71], statistical analysis [22, 67], or learning models [10, 33, 82]). The key idea of these approaches is to identify the most recurrent fix actions. However, as the repetitiveness of fixing changes decreases exponentially as the code change size grows [51], the ranking of complicated fixing changes degrades and thus has a low probability to be chosen as fix actions. Hence, such complicated changes are hard to be crafted or learned. For example, Figure 2 presents a typical bug named Chart-6 from Defects4J that cannot be fixed by existing APR techniques. The fix action of Chart-6 comprises multiple insertions of different

```
@@ -108,7 +108,14 @@
   if (!(obj instanceof ShapeList)) {
     return false;
   }
-  return super.equals(obj);
+  ShapeList that = (ShapeList) obj;
+  int listSize = size();
+  for (int i = 0; i < listSize; i++) {
+    if (!ShapeUtilities.equal((Shape) get(i), (Shape) that.
↪  get(i))) {
+      return false;
+    }
+  }
+  return true;
```

Figure 2: Chart-6 bug from Defects4J that has not been correctly fixed by any APR technique yet.

control structures, including assignments, branches, loops, and return statements. Such actions are still out of the reach of existing APR techniques.

The third challenge is ***how to adapt complicated donor code***. Existing APR techniques have proposed several approaches to transfer donor code to buggy locations. GenProg [66] directly transfers code from one location within the buggy program to the buggy location without solving potential namespace conflicts. The subsequent APR techniques (e.g., SimFix [22], SOSRepair [5], ssFix [71]) have made significant progress by proposing variable renaming approaches to adapt donor code snippets. But when the donor code contains more entities (e.g., user-defined data types) other than variables, a more comprehensive donor code transfer approach may be desirable. For example, as shown in Figure 2, fixing such a bug generally requires a set of new code fragments. Apart from multiple variables, the Chart-6 patch contains both primitive and user-defined data types, and even method calls from different classes. Furthermore, the combination of these donor codes spans up to eight-line insertions at a hunk.

In this paper, we present a novel graph differencing based code transplantation approach to help address the three challenges. The concept of code transplantation, which aims to transfer code from the donor location to the host location, is proposed by Barr et al. [8]. As a promising approach to reuse code, code transplantation opens an avenue for automated program repair [44]. Specifically, in this paper, we choose to search fix ingredients at the method level within the buggy project, and propose an inheritance-hierarchy-aware code search approach that leverages the inheritance relations to effectively locate fix ingredients (**the first challenge**). Then, we represent methods via attributed control flow graphs and reformulate the fix action extraction as a graph edit computation problem. We further propose to use a graph differencing approach to distill semantic fix actions (**the second challenge**). As the donor code residing in the donor method cannot be directly applied to the host due to different variables, types, and methods, we propose a novel namespace transfer approach to effectively adapt the donor code to the buggy method (**the third challenge**).

In summary, the major contributions of this work include:

[1]More detail of the review can be found at Section 4.3.

(1) An inheritance-hierarchy-aware method-level donor search approach, which captures both local feature (i.e., method signatures) and global feature (i.e., inheritance relations) of methods to effectively locate donor method within the buggy project.
(2) A graph differencing based approach to distill semantic fix actions. We reformulate the fix action extraction as a graph edit distance computation problem, and distill fix actions from the edit graph between the buggy and donor method.
(3) A comprehensive namespace construction approach that considers the transfer of not only variables but also data types and methods, which enables larger donor code adaption.
(4) *TransplantFix* brings new momentum to help break through the bottleneck in the growth of newly fixed bugs, by correctly fixing eight new bugs that have never been fixed by prior APR techniques. This is the best performance as compared to the state-of-the-art APR techniques proposed in the last three years. *TransplantFix* has also shown the best generalizability among all APR techniques evaluated on Defects4J v1.2 and v2.0. In addition, *TransplantFix* is found to have the capability to synthesize correct patches consisting of at most eight-line insertions (e.g., Chart-6).

To facilitate follow-up research, we make our prototype and all relevant artifacts publicly available at https://github.com/DehengYang/TransplantFix.

2 MOTIVATING EXAMPLE

```
103 public boolean equals(Object obj) {
104     if (obj == this) {
105         return true;
106     }
107     if (!(obj instanceof ShapeList)) {
108         return false;
109     }
110 
111     return super.equals(obj);
112 }
```

Figure 3: The buggy method in `ShapeList` class of Chart-6.

In this section, we present a real-world bug example that motivates our approach. Figure 3 lists the buggy method of Chart-6 that cannot be fixed by prior state-of-the-art APR techniques.

We obtained insights for addressing **the first challenge** (i.e., where we can obtain such fix action and donor code) by taking a closer look at the whole Chart-6 buggy program. We observed a donor method that could potentially provide the two sources of fix ingredients (i.e., fix action and donor code) for Chart-6. As shown in Figure 4, the donor method `equals()` in `PaintList` class shares the same method name as the buggy method of Chart-6. Instead of invoking `super.equals(obj)` as the buggy method does (see line 111 in Figure 3), the donor method has its own implementation for comparison (see lines 98-107 in Figure 4). Such implementation in the donor method is functionally similar to that in the human-written patch of Chart-6 as shown in Figure 2. Interestingly, both the buggy class (i.e., `ShapeList`) and donor class (i.e., `PaintList`) inherit the same parent class (i.e., `AbstractObjectList`). This points out a new avenue to obtain the complicated fix ingredients (e.g., Chart-6) by properly searching donor methods inside the buggy project.

```
91  public boolean equals(Object obj) {
92      if (obj == null) {
93          return false;
94      }
95      if (obj == this) {
96          return true;
97      }
98      if (obj instanceof PaintList) {
99          PaintList that = (PaintList) obj;
100         int listSize = size();
101         for (int i = 0; i < listSize; i++) {
102             if (!PaintUtilities.equal(getPaint(i),
                ↪   that.getPaint(i))) {
103                 return false;
104             }
105         }
106     }
107     return true;
108 }
```

Figure 4: One donor method in `PaintList` class of the Chart-6 buggy project.

Targeting this problem, we design an inheritance-hierarchy-aware method search approach. Considering that the buggy method and donor method of Chart-6 are significantly different in terms of identifiers, control structures, and patch size, we choose to use the method signature to measure the similarity between the two methods. However, when the method signature is very common in the project (e.g., `equals()`), the candidate donor methods may have the same similarity scores (i.e., being "tied"), which is undesirable for donor method ranking. To break the ties, we further leverage the inheritance relations by prioritizing the donor methods that lie in the same inheritance graph of the buggy method (Section 3.2). In this way, our approach accurately ranks the donor method at the Top-1 position.

To address **the second challenge** (i.e., how to obtain complicated fix actions), we propose to perform graph differencing to extract the semantic differences between the buggy and donor method. We first represent the method as a control flow graph, and then design a graph differencing algorithm based on an exact graph edit distance algorithm [4] to infer all possible fix actions. In this example, our algorithm obtains an edit graph by differencing the control flow graphs constructed from the buggy and donor method. As shown in Figure 5, the edit graph specifies semantic fix actions between the buggy and donor method, which comprises a search space of fix actions. This search space includes the fix action (i.e., updating the statement at line 111 in the buggy method with statements ranging from lines 98-107 in the donor method) needed for fixing Chart-6.

Even when the fix action is obtained, it is still challenging to adapt the donor code to the buggy method (**the third challenge**). As shown in Figure 4, the data types (e.g., `PaintList`) and method calls (e.g., `PaintUtilities.equal()`) are not compatible with the buggy method. Such adaption is out of the scope of existing code adaption approaches used in APR techniques [5, 22, 71]. To address the third challenge, we devise a namespace transfer approach that could deal with not only variables but also types and methods. This enables us to successfully infer the `ShapeUtilities.equal()` that

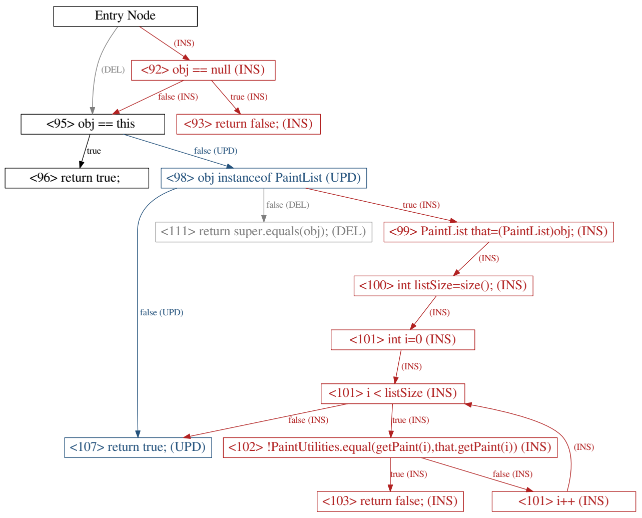

Figure 5: The edit graph of Chart-6 generated by our graph differencing approach. Black boxes represent matched nodes, gray boxes mean nodes to be *deleted*, blue boxes denote nodes to be *updated*, and red boxes denote nodes to be *inserted*.

does not exist in the buggy file but is in fact compatible with the buggy file. After correctly transferring the donor code, we can apply the extracted semantic fix action and the transferred donor code to the buggy method to produce a correct fix for Chart-6.

3 APPROACH

In this section, we first present an overview of *TransplantFix* in Section 3.1. We then illustrate the three core algorithms, i.e., fix ingredient search, donor method transfer, and fix action extraction, in Section 3.2, Section 3.3, Section 3.4, respectively.

3.1 Overview

Figure 6 depicts the overall workflow of *TransplantFix*. *TransplantFix* builds on top of the typical pipeline of automated program repair that consists of three stages: 1) fault localization; 2) patch generation; and 3) patch validation. For fault localization, *TransplantFix* takes the buggy program and its associated test suite as input and produces a ranking list of suspicious methods in descending order of their suspiciousness values. This list is then fed into the patch generation stage to produce candidate patches. As the core stage of *TransplantFix*, the patch generation stage is comprised of three phases: 1) fix ingredient search; 2) donor code transfer; and 3) fix action extraction. For each suspicious method, the fix ingredient search phase measures its similarity to all methods inside the project, which are then ranked based on the similarity. For each donor method, the donor code transfer phase translates the entities in the donor method into entities that are compatible with the suspicious method. After that, the fix action extraction phase represents the methods with attributed control flow graphs, and produces the edit graph by differencing the transferred donor graph and the suspicious graph. Based on the edit graph, it extracts the semantic fix actions to construct a search space. It applies fix actions in the search space with concrete donor code to the suspicious method to produce a list of candidate patches. In the last stage, patch generation validates the candidate patches with the original test suite of the buggy program. If no valid patch that passes the test suite is found, *TransplantFix* goes back to the patch generation stage for the next suspicious method. Otherwise, *TransplantFix* exits and outputs the valid patch.

3.2 Fix Ingredient Search

Fix ingredient search aims to distinguish the donor method that contains fix ingredients for bug fixing from other methods. To this end, it leverages both local feature (i.e., method signature) and global feature (i.e., inheritance relations) to measure the functional similarity between the suspicious method and each candidate donor method. First, we present definitions related to the local and global features of methods as follows.

Definition 1 (Method Signature). Given a method m, a *method signature* is a pair: $\langle MethName, ArgTypeList \rangle$. The first element is the method name, and the second element is a list of argument types of the method, i.e., $[ArgType_1, ArgType_2, ..., ArgType_n]$.

We define the method signature according to the Java Language Specification (JLS for short) [2]. The JLS specifies that two methods are considered to have the same signature if they have the same method name and argument types, regardless of the return types and argument names. Therefore, Definition 1 characterizes the method signature by method name and a list of argument types.

Definition 2 (Inheritance DAG). The *inheritance DAG* I is a directed acyclic graph (DAG for short) representing inheritance relations among classes. Each vertex in I denotes a class, and the edge from $vertex_i$ to $vertex_j$ denotes that $class_i$ is inherited from $class_j$.

Inheritance is an important feature of object-oriented programming, as a mechanism to enable one class to inherit the fields and methods of another class. It is also reported that about 74% of user-defined classes use inheritance in at least half of the investigated real-world Java applications [63]. As a class can extend at most one super class but implement multiple interfaces in Java, we adopt graph structure (i.e., inheritance DAG as shown in Definition 2) rather than tree structure to represent inheritance relations. This is also adopted by previous inheritance-related studies [63]. We use *IDAGs* to denote the set of inheritance DAGs.

Definition 3 (Relative Class). *$class_i$ is a relative class of $class_j$ iff $\exists I \in IDAGs, class_i \in I.vertices \land class_j \in I.vertices$.*

We propose the concept of "relative class" to describe the relation of two classes in the same inheritance DAG. The relative class captures the global feature of methods, and thus can help further prioritize donor methods. For example, the donor class `PaintList` is a relative class of the buggy class `ShapeList` in Chart-6, and the donor method will be given a higher score when measuring the similarity.

Given two methods m (from buggy code) and m' (from donor code), we measure their similarity based on the local feature (i.e., method signature) as follows:

$$\begin{aligned} Sim_{local}(m, m') = \lambda_1 * jacc(MethName(m), MethName(m')) \\ +\lambda_2 * jacc(ArgTypeList(m), ArgTypeList(m')) \end{aligned} \quad (1)$$

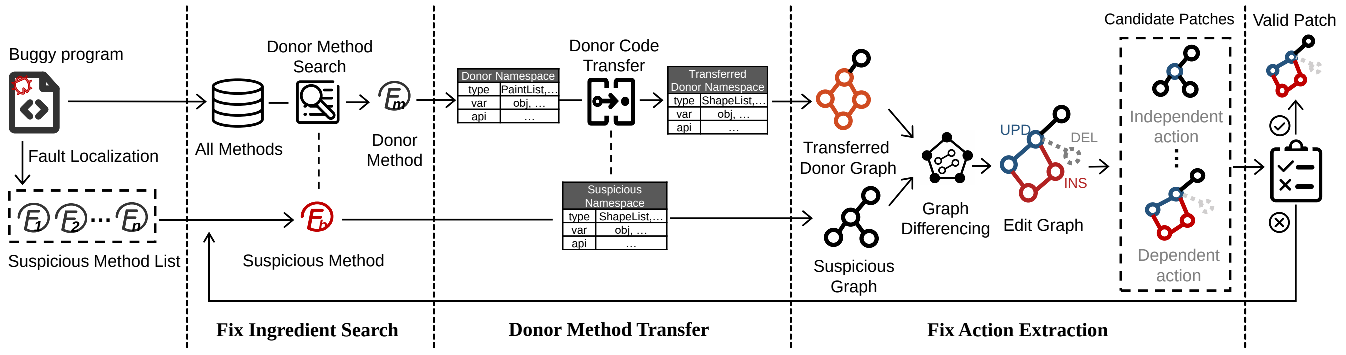

Figure 6: The overview of *TransplantFix*.

As shown in Equation (1), we include two elements of method signature (i.e., method name and the argument type list), and adopt Jaccard similarity coefficient[2] (i.e., `jacc()`) to compute the similarity of the two local features separately. As the naming convention in Java generally conforms to CamelCase, we split the method name into subtokens, and then feed the subtokens sets of two methods into `jacc()`. We perform the same splitting operation on argument types. λ_1 and λ_2 represent the weights assigned to the similarity score of method name and argument type list, respectively.

We measure the similarity of the global feature (i.e., inheritance relations) as follows:

$$Sim_{global}(m, m') = \begin{cases} 1 & \text{if } relative(cls(m), cls(m')) \\ \mu_1 * jacc(PkgName(m), PkgName(m')) + \\ \mu_2 * jacc(ClsName(m), ClsName(m')) & otherwise \end{cases} \quad (2)$$

In Equation (2), we assign the similarity score as 1 when the classes of the two compared methods are relative classes (i.e., *relative*($cls(m), cls(m')$)). For the cases when there are no relative classes for the buggy class, we further consider the similarity between two elements, i.e., *PkgName* as the package name and *ClsName* as the class name. For the two elements, we use the same similarity measure (i.e., Jaccard) and splitting operation to obtain a similarity score. Likewise, μ_1 and μ_2 denote the weights assigned to the similarity scores of package name and class name, respectively.

We finally merge the Sim_{local} and Sim_{global} by computing their average to obtain the overall similarity score between two methods. In this way, we consider both local features and global features to rank the candidate donor methods. Note that there are four weight parameters in the similarity measure equations. As there is still no work studying on how an exact value each weight should be assigned, in this work, we straightforwardly assign all parameters (i.e., λ_1, λ_2, μ_1, and μ_2) as 0.5. We finally obtain a ranking list of the top 50 candidate donor methods in descending order of the overall similarity score to the suspicious method.

3.3 Donor Method Transfer

Donor method transfer aims to align the donor namespace with the suspicious namespace. It first constructs the namespace for the suspicious method and the candidate donor method by collecting three types of program entities in each method. Then, it creates entities mappings that encode how entities in the candidate donor method should be adapted to fit for the suspicious method.

3.3.1 Namespace construction. Before transferring the namespace of the candidate donor method, it is indispensable to construct the namespace for both the suspicious method and candidate donor method. For a method, the namespace construction covers the following three kinds of program entities:

(1) Types: we consider two kinds of types. The first is all types used in the method, serving as the "local types". The second is the corresponding suspicious class type that the method belongs to (e.g., `ShapeList` for the suspicious method `equals()` as shown in Figure 3), serving as the "global type".
(2) Method calls: we collect all method calls invoked in the method. To facilitate the comparison between two method calls, we identify and collect its local feature (i.e., method signature) as well as global feature (i.e., inheritance relations).
(3) Variables: apart from collecting each variable defined or used in the method, we further collect its type by tracing its declaration.

3.3.2 Namespace Alignment. Once the two namespaces are obtained, we further build a namespace mapping that covers types, method calls, and variables. As both variables and method calls in the namespace are related to type information, we prioritize to construct type mapping to facilitate mappings of other entities.

Type mapping. We first build mappings for the "global type". Given a "global type" of the suspicious method *type* and the other of the candidate donor method *type'*, we split *type* into a list of subtokens based on the CamelCase naming convention, i.e., $list = [tk_1, tk_2, ..., tk_m]$. Likewise, we split $type'$ into $list' = [tk'_1, tk'_2, ..., tk'_n]$. We then compute the longest common subsequences (LCS) between the two lists. Finally, we build a function $f_1 \subset (S' \cup concat(S'_{sub})) \times (S \cup concat(S_{sub}))$, where $S = list$, $S' = list'$, S_{sub} (and S'_{sub}) means the set of subsets of S (and S'), to map a subtoken or concatenation of subtokens in $list'$ to that in $list$. Let tk'_i, tk'_{i+k} be two neighbour elements in LCS of the two lists. We define $f_1(tk'_i) = tk_j$ where $tk_j = tk'_i$, $f_1(tk'_{i+k}) = tk_{j+r}$ where $tk_{j+r} = tk'_{i+k}$, $f_1(concat(tk'_i, \ldots, tk'_{i+k})) = concat(tk_j, \ldots, tk_{j+r})$. Taking the donor method of Chart-6 in Figure 4 as an example, the LCS of `PaintList` and `ShapeList` is $[List]$, and the unique subtoken Paint in PaintList is mapped to the unique subtoken Shape in ShapeList. Thus, $f_1(Paint) = Shape$.

[2]https://en.wikipedia.org/wiki/Jaccard_index

Based on the global type mapping f_1, We then construct the local type mapping f_2. For each type in the candidate donor method, if it contains S in $dom(f_1)$, we substitute the S in the type with $f_1(S)$. Otherwise, we leave it unchanged. For example, the local type mapping for the donor method of Chart-6 is $f_2(PaintList) = ShapeList$, $f_2(PaintUtilities) = ShapeUtilities$.

Method call mapping. We define the method call mapping with a function $f_3 \subseteq M' \times M$, where $M' = \{m'_1, m'_2, ..., m'_i\}$, $M = \{m_1, m_2, ..., m_j\}$. M' and M represent the set of method calls in the candidate donor method and suspicious method respectively. We first map method calls from the candidate donor method to those in the suspicious method by measuring their similarity. We use the same measurement described in Section 3.2 to measure the similarity between two method calls. When the similarity is larger than the threshold score, we add the two method calls into the mapping. We set the threshold as 0.6 in this paper.

For method calls that still have no mappings, we further create method call mappings based on the global type mapping. Similar to the construction of local type mapping, only when the method call contains one S in $dom(f_1)$, do we substitute the S in the method call with $f_1(S)$. For instance, the method call mapping for the donor method of Chart-6 is $f_3(getPaint) = getShape$.

Variable mapping. The variable mapping f_4 is built on top of the type mapping. Specifically, for each variable var_i in the candidate donor method, we include a set of variables $vars$ in the suspicious method that conform to the following constraint: $\forall var_j \in vars, type(var_j) = type(var_i) \vee type(var_j) = f_2(type(var_i))$, where $type(var)$ means the type of the variable. We further rank variables in $vars$ in ascending order of the Levenshtein distance to the var_i. For example, the variable mapping for Chart-6 donor method is $f_4(obj) = \{obj\}$.

3.4 Fix Action Extraction

Fix action extraction aims to extract the semantic fix actions between the candidate donor method obtained in Section 3.2 and the suspicious method, based on the transferred namespace of the candidate donor method in Section 3.3. Specifically, it consists of the following steps:

3.4.1 Graph Construction. As discussed in Section 1 and Section 2, when the fix action involves complex control structures, it is often out of reach of existing APR techniques. Therefore, we propose to represent methods via control flow graphs (CFGs for short), as a CFG could model the control dependencies between statements in a method [6] and facilitate the extraction of semantic fix actions.

We adopt Soot [65], an open-source and well-maintained tool that has been extensively used for various program analyses, to obtain the CFG for each method. Soot has also been used in prior APR techniques [26, 62, 74]. Note that Soot produces a CFG of methods by analyzing the bytecode of a class, but our approach generates candidate patches by modifying the Abstract Syntax Tree (AST) at the source-code level. Therefore, we further translate the CFG produced by Soot to a CFG in which each node corresponds to an AST node. In the translated CFG, the branch nodes (e.g., the if expression) is represented at the expression AST level, and other nodes are represented at the statement AST level. Such CFGs are then fed into the graph differencing algorithm to infer semantic fix changes.

3.4.2 Graph Differencing. Based on the constructed CFGs, we reformulate the fix change extraction as a graph edit distance computation problem. We present related definitions as follows:

Definition 4: (Attributed CFG). An attributed CFG is a tuple with four elements: $\langle V, E, \mu, \xi \rangle$, where V and E denote a set of vertices (i.e., AST nodes) and edges (i.e., flow transitions), respectively. $\mu : V \longrightarrow A_V$ is a function that associates attribute a_e to a vertex, and $\xi : E \longrightarrow A_E$ is a function that assigns attribute a_e to an edge.

As shown in Definition 4, we represent AST nodes with vertices and flow transitions with edges in an attributed CFG. In terms of the attributes, we use the formatted string of each AST node in the CFG as the vertex attribute, and associate the flow transition label (i.e., "true" and "false" for branch flow transition, and the empty string for other transitions) with the edge attribute.

Definition 5: (Graph Edit Distance). Given two attributed CFGs: $g : \langle V, E, \mu, \xi \rangle$, $g' = \langle V', E', \mu', \xi' \rangle$, the graph edit distance from g to g' is:

$$GED(g, g') = \min_{(eo_1, \ldots, eo_k) \in \varphi(g, g')} \sum_{i=1}^{k} c(eo_i)$$

where eo_i denotes an edit operation between two vertices or two edges, and $c(eo_i)$ means the cost of the edit operation eo_i. $(eo_1, ..., eo_k)$ represents an edit path (i.e., a sequence of edit operations), and $\varphi(g, g')$ represents all possible edit paths from g to g'.

Graph edit distance (GED) is a graph differencing approach that aims to find the best edit path from $\varphi(g, g')$ with minimum cost. As shown in Definition 5, eo_i is an edit operation for two vertices or two edges, and there are four types of edit operations: 1) **update**: $\{v \longrightarrow v'\}$ means updating v with v'; 2) **deletion**: $\{v \longrightarrow \varepsilon\}$ means deleting v; 3) **insertion**: $\{\varepsilon \longrightarrow v'\}$ means inserting v'; 4) **match**: *match* operation is a special case of *update* operation, but its cost is zero (i.e., $c(eo_i) = 0$), indicating that v and v' are exactly matched. The edit operation for edges shares the same types as these of vertices.

With the above definitions, we have reformulated the fix action extraction as a graph edit distance computation problem. To solve the problem, we adopt the DF-GED algorithm, a novel GED approach proposed by Abu et al. [4], and we further design the cost function to compute the cost of each pair of vertices and each pair of edges:

$$Cost(v, v') = \begin{cases} \min\limits_{A_{v'_j} \in A_{v'}} levensh(A_v, A_{v'_j}) & \text{if typeMatched}(v, v') \\ maxCost & otherwise \end{cases} \tag{3}$$

where A_v is the attribute of vertex v in g, and $A_{v'}$ denotes the attribute of vertex v' in g'.

$$Cost(e, e') = (levensh(A_e, A_{e'}) + Cost(A_{v_e^{in}}, A_{v_{e'}^{in}}) + Cost(A_{v_e^{out}}, A_{v_{e'}^{out}}))/3 \tag{4}$$

where A_e is the attribute of edge e. $A_{v_e^{in}}$ is the attribute of the start vertex of e, and $A_{v_e^{out}}$ is the attribut of the end vertex of e.

As shown in Equation (3), if both vertices are expressions or both are statements (i.e., *typeMatched()*), we compute the Levenshtein distance (i.e., *levensh()*) for the string of v and each of mapped strings of v' obtained in Section 3.3. We normalize each Levenshtein distance into the range [0, 1], and select the least cost as the final cost. Otherwise, we regard the two vertices are not matched, and thus assign *maxCost*. Note that the *maxCost* should be larger than the cost of matched nodes, we set it as 10 in this work.

The cost computation of two edges builds on top of that of vertices. As shown in Equation (4), we compute the average cost of the Levenshtein distance between the label of e and e', the cost between the start vertices (i.e., $v_e^{in}, v_{e'}^{in}$) of the two edges, and that between the end vertices (i.e., $v_e^{out}, v_{e'}^{out}$) of the two edges.

3.4.3 Search Space Construction. Note that the graph edit distance algorithm in Section 3.4.2 finally outputs a best edit path (i.e., a sequence of edit operations $(eo_1, ..., eo_k)$), which contains all edit operations needed to transform g to g'. However, in the context of automated program repair, the buggy method may only need a subset of the edit operations for fixing the bug, and introducing all edit operations may lead to side effects or errors. On the other hand, the search space could explode if considering all subsets of the set of edit operations involved in the best edit path. For example, as shown in Figure 5, there are 12 edit operations for vertices (i.e., nine insertions, two updates, and one deletion), which correspond to a search space of $2^{12} = 4,096$ candidate patches. The search space could grow up to a million candidate patches when there are over 20 edit operations. To effectively shape the search space, we propose the notion of "cut point" and use cut point pairs to group fix actions. After grouping, each group can be considered as one "large" edit operation.

Definition 6 (Cut Point). A cut point is an edit operation for two vertices: $eo^c = v \longrightarrow v'$, where vertex $v \in g$, vertex $v' \in g'$, and eo^c is of *update* or *match* type.

As shown in Definition 6, a cut point is defined as an *update* or edit operation, where the two vertices exist in the two graphs. A cut point indicates a mapping between one vertex in the suspicious method and one vertex in the candidate donor method, and can be further leveraged to group dependent edit operations.

Algorithm 1 describes how we group edit operations and generate fix actions. We first get all cut points by performing a depth-first search on g_s (line 5). Then, we leverage a pair of cut points, i.e., (eo_i^c, eo_j^c), where $eo_i^c = \{v_i \rightarrow v_i'\}, eo_j^c = \{v_j \rightarrow v_j'\}$, to cut the whole edit path into pieces (lines 6-14). In this way, we can characterize a sequence of continuous edit operations that lie between the eo_i^c and eo_j^c. These continuous operations can be combined as one "large" dependent fix action (line 13). When identifying the pair of cut points, we further introduce a line order constraint that encodes syntactic rules: $v_i.lineNo < v_j.lineNo \wedge v_i'.lineNo < v_j'.lineNo$, where *lineNo* indicates the line number of the corresponding vertex. This requires that the two vertices of the eo_i^c and eo_j^c in the suspicious method are supposed to have the same line number order as those in the candidate donor method (lines 8-11). For generating fix actions (line 13), apart from obtaining a "large" fix action considering the complete sequence of edit operations, we further enrich the search space by generating fix actions that apply one single edit of the sequence. Once all fix actions are generated from cut point pairs, we further consider to combine "large" fix actions that lie in the same if or loop control structure to form the branch-dependent fix actions (line 16). With these actions, we could then apply them one by one with the concrete adapted donor code to the buggy project to generate a list of candidate patches.

Algorithm 1 The cut-and-group algorithm to extract fix actions.

```
Input: The edit path, $ep : (eo_1, ..., eo_k)$
Input: The suspicious CFG, $g$
Input: The candidate donor CFG, $g'$
Output: Fix actions, $actions$
1:  function CUTANDGROUP($ep, g, g'$)
2:      // Group
3:      $actions \leftarrow \emptyset$
4:      $visited \leftarrow \emptyset$
5:      $cutPoints = DFS(g, ep)$
6:      for $eo_i^c \in cutPoints \wedge eo_i^c \notin visited$ do
7:          $eo_j^c \leftarrow cutPoints.nextCutPoint(eo_i^c)$
8:          while $violateLineOrderConstraint(eo_i^c, eo_j^c, g, g')$ do
9:              $visited \leftarrow visited \cup eo_j^c$
10:             $eo_j^c \leftarrow cutPoints.nextCutPoint(eo_j^c)$
11:         end while
12:         // Generate
13:         $actions \leftarrow actions \cup generateActions(eo_i^c, eo_j^c, g, g')$
14:     end for
15:     // Further include branch-dependent actions
16:     $branchDepActions \longleftarrow groupActsInSameIfOrLoop(actions)$
17:     return $actions \cup branchDepActions$
18: end function
```

4 EVALUATION

4.1 Implementation

We have implemented *TransplantFix* for Java programming language in about 20.4 KLoc of source code. Apart from the core modules described in Section 3, *TransplantFix* includes the fault localization and patch validation modules to form a complete repair pipeline. For fault localization, we leverage the GZoltar API [9] and Ochiai similarity coefficient [3], which are commonly used in existing APR tools [22, 35, 67, 75, 82], to implement a method level fault localization. For patch validation, we first run bug failing test cases on the candidate patch. If all failing test cases pass, we then run a regression test by executing all positive test cases on the patch. For plausible patches that pass the test suite, we manually assess the correctness of these patches by checking if they are identical or semantically equivalent to the human-written patches. We set the time budget for each repair trial as two hours, following the prior practice in the APR community [15].

4.2 Dataset

We measure the effectiveness of *TransplantFix* on the Defects4J v1.2 dataset [24], and further evaluate the generalizability of *TransplantFix* on the Defects4J v2.0 dataset. Defects4J v1.2 is an extensively used real-world bug dataset where tens of APR tools have been evaluated [15, 37, 40]. It consists of 395 bugs from six open source projects, i.e., jfreechart, commons-lang, commons-math, closure-compiler, joda-time, and mockito. The size of these buggy programs ranks from 22 to 96 KLoc.

To alleviate the benchmark overfitting problem reported by Durieux et al. [15], we adopt the extended version of Defects4J, i.e., v2.0, to further measure *TransplantFix*, which is also used in the literature to evaluate state-of-the-art APR techniques [25, 78, 82]. Defects4J v2.0 covers 17 real-world software systems and consists of 835 real-world bugs [17], including 444 new bugs as compared to v1.2. To avoid repeated repair trials on the same bug, we evaluate *TransplantFix* on the 444 new bugs in Defects4J v2.0. The full bug list and repair data in our evaluation are also publicly available [1].

4.3 Effectiveness of *TransplantFix*

In this section, we evaluate the effectiveness of *TransplantFix* on the 395 bugs from Defects4J v1.2, and compare it with existing APR techniques.

Since the patch overfitting problem is observed [53, 61], assessing the correctness of APR-generated patches has become a consensus in the literature. Accordingly, the number of correctly fixed bugs ($N_{correct}$ for short) has been widely evaluated to show the effectiveness of APR techniques [22, 33, 37, 42]. In recent years, the number of bugs that are correctly fixed for the first time (N_{unique}) has been an increasingly popular metric to emphasize the unique contributions of an APR techniques, and has been extensively adopted by the literature [12, 22, 57, 78, 82]. As the number of APR techniques grows continuously, how to correctly fix more bugs for the first time is a challenging task for the APR community.

In this paper, we focus on measuring N_{unique} of *TransplantFix* against existing APR techniques and perform a thorough comparison by including 42 APR techniques. To conduct such a comparison, we conducted a systematic literature review to identify all APR tools that have been evaluated on Defects4J dataset. We consider four inclusion criteria during the review: 1) the paper presents a new APR technique and evaluates it on Defects4J, e.g., APR tools targeting C language are excluded; 2) the paper must be peer-reviewed, i.e., we exclude grey literature [69] including technical reports, Ph.D dissertations, etc.; 3) the full-text of the paper ought to be publicly accessible to enable our check on which bugs are fixed by this APR tool. For example, DEAR [34] which still has no publicly available full-text at the time of our literature review[3] is excluded.

Based on such inclusion criteria, we finally identified 42 APR tools. As shown in Table 1, our comparison covers a seven-year time period (i.e., from 2016 to 2022), and characterizes the differences in the fault localization configuration among APR techniques. We observed that existing evaluation of APR techniques mainly employ two fault localization configurations:

(1) Fault Localization (**FL** for short) that deploys a specific fault localization technique (i.e., spectrum-based [22, 37] or information retrieval-based [30]) and outputs a ranking list of suspicious program entities to the APR tool;
(2) Non-Fault Localization (**Non-FL** for short) that straightforwardly provides the faulty lines [10, 33] or methods [31] to the APR tool.

[3]The literature review was performed on Apr 12, 2022.

Table 1: The 42 APR tools included in our comparison with *TransplantFix*.

Year	#	APR Tools
2016	6	DynaMoth [16], jGenProg [46], jKali [46], jMutRepair [46], Nopol [75], HDRepair† [31]
2017	4	ssFix [71], JAID [11], ACS [72], Elixir [55],
2018	10	Cardumen [47], SimFix [22], CapGen [67], SketchFix [20], RSRepair-A [79], Kali-A [79], GenProg-A [79], SOFix [41], Arja [79], LSRepair [38],
2019	11	Avatar [36], DeepRepair [68], Hercules [57], GENPAT [21], ConFix [28], Vfix [74], PraPR [18], iFixR [30], SequenceR† [13], kPAR‡ [35], TBar‡ [37]
2020	7	Arja-e [80], Restore [73], JAID-extend [12], FixMiner [29], CODIT† [10], DLFix†[33], CoCoNut†[42]
2021	3	VarFix [70], Recoder‡ [82], CURE†[23]
2022	1	RewardRepair‡ [78]

# means the number of APR tools proposed in each year. † means the APR tool is evaluated in Non-FL scenario (i.e., the faulty lines or methods are directly given), and ‡ indicates the APR tool is evaluated in both FL scenario and Non-FL scenario. For the rest of APR tools, they are all evaluated in FL scenario.

As the two fault localization scenarios have significantly different impacts on the effectiveness of APR tools [35, 40], we compare the two fault localization scenarios separately. Following prior studies [22, 37, 78, 82], we collect repair results of the 42 investigated tools from the literature.

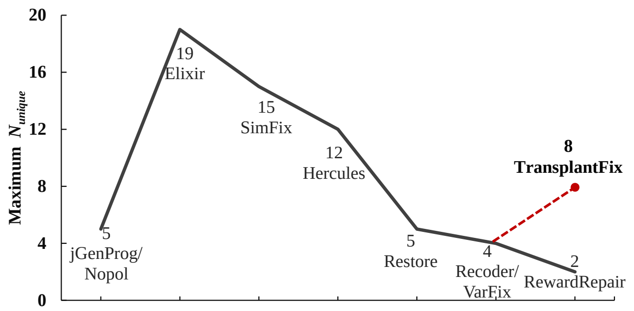

Figure 7: The maximum N_{unique} achieved on Defects4J v1.2 in each year with FL scenario. Apart from *TransplantFix*, 36 APR techniques using the FL scenario are considered.

[Tendency of N_{unique} in the last seven years.] Figure 7 presents the maximum N_{unique} in each year in FL scenario. For each APR tool, N_{unique} is the number of bug fixes that have never been fixed by prior APR tools proposed before the year when the APR tool is created. For example, Elixir [55] is proposed in 2017, and thus we compute its N_{unique} by checking how many new bugs are not fixed by the five APR tools (i.e., in the FL scenario, HDRepair is excluded as it relies on Non-FL scenario) proposed in 2016 but fixed by Elixir. Accordingly, the maximum N_{unique} is the highest N_{unique} achieved each year. In 2017, the N_{unique} of Elixir is 19, which is higher than the other three tools proposed in 2017. Therefore, the maximum N_{unique} is 19 in 2017.

As indicated in Figure 7, the growth of newly fixed bugs in the dataset has hit a bottleneck in recent years. The maximum N_{unique} rapidly rises to 19 in 2017, and steadily declines in the next two years (i.e., 2018 and 2019). The maximum N_{unique} suddenly shrinks into five and four in 2020 and 2021. Furthermore, this number drops into two in 2022, without considering *TransplantFix*. *This indicates that, as a number of APR techniques have been continuously proposed, fixing new bugs is becoming increasingly difficult.*

[Effectiveness of *TransplantFix* in FL scenario.] In Figure 7, 36 APR tools that are evaluated in FL scenario have been considered, and a total of 146 bugs in Defects4J v1.2 have been correctly fixed by at least one of the 36 tools. To compute the N_{unique} of *TransplantFix*, we not only consider all fixed bugs between 2016 and 2021, but also include the newly fixed bugs achieved by RewardRepair in 2022. Accordingly, *TransplantFix* correctly fixes eight new bugs that have not been fixed by any of the 36 APR techniques. Furthermore, as compared to the number of newly fixed bugs of the seven APR tools proposed in the last three years and evaluated in FL scenario, *TransplantFix* achieves the best performance, which is 60% (3 bugs) more than Restore and 300% (6 bugs) more than RewardRepair.

[Effectiveness of *TransplantFix* in Non-FL scenario.] Using the Non-FL scenario has been a common practice since 2019, especially among deep learning based APR tools (i.e., 8 out of 12 tools in Table 2). Table 2 lists the N_{unique} of 12 APR tools, including *TransplantFix*, which employ the Non-FL scenario. As one of the pioneer tools that performed evaluation in Non-FL scenario, TBar fixed 64 bugs for the first time as compared to HDRepair. After 2019, the N_{unique} has never exceeded 20. In 2021, Recoder achieved a relatively high N_{unique} as compared to all (i.e. seven) tools proposed between 2016 and 2020. The difficulty of repairing new bugs has become larger in 2022. One representative fact is that, the state-of-the-art APR technique proposed in 2022, i.e., RewardRepair, has fixed only 5 bugs for the first time, respectively. We compute the N_{unique} of *TransplantFix* based on all bugs that are correctly fixed by APR tools evaluated in Non-FL scenario, including the two APR tools proposed in 2022. *In such a situation, TransplantFix has still successfully fixed 10 new bugs that have never been correctly fixed by all 10 prior APR tools.*

Table 2: The N_{unique} achieved on Defects4J v1.2 in each year with Non-FL scenario. Apart from *TransplantFix*, 10 APR techniques evaluated in the Non-FL scenario are included.

Year	Tool	N_{unique}	Year	Tool	N_{unique}
2016	HDRepair	6	2020	CoCoNut	12
2019	KPar	33	2021	CURE	0
2019	TBar	64	2021	Recoder	16
2019	SequenceR	14	2022	RewardRepair	5
2020	CODIT	4	2022	*TransplantFix*	10
2020	DLFix	8			

4.4 Generalizability of *TransplantFix*

Since Durieux et al. revealed the benchmark overfitting problem [15], evaluating the generalizability of a newly proposed APR tool has been becoming increasingly popular. In this work, we evaluate the generalizability of *TransplantFix* on the 444 new bugs in Defects4J v2.0. Note that existing APR tools evaluated on this dataset use different fault localization scenarios. For a fair comparison, we run *TransplantFix* in both FL and Non-FL scenarios. Table 3 presents the number of correctly fixed bugs achieved by each APR tool.

We observed that prior evaluation of the generalizability of an APR tool mainly focuses on the absolute number of correctly fixed bugs [25, 82], but this may introduce threats to the reliability of the conclusion. For example, an APR technique correctly fixes 20 bugs in 395 bugs from Defects4J v1.2, and 20 bugs from 444 bugs in Defects4J v2.0. There will be no performance degradation in terms of the absolute number, but actually the performance of the tool slightly degrades when considering the differences between the dataset size (i.e., 395 vs. 444). To capture such differences, we propose the following metric to compute the generalizability of an APR technique:

$$DegradationRatio = \frac{N_{correct}}{|usedBugs|} - \frac{N'_{correct}}{|usedBugs'|} \tag{5}$$

where $N_{correct}$ means the number of correctly fixed bugs in the original dataset, and $N'_{correct}$ means the number of correctly fixed bugs in the new dataset. $|usedBugs|$ and $|usedBugs'|$ represent the number of used bugs for evaluation from the original dataset and new dataset, respectively. The lower the *DegradationRatio* score, the more generalizable the APR tool.

Table 3: The *DegradationRatio* score of each APR tool.

Tool	v1.2	v2.0	*DegradationRatio*
SimFix (FL)	34/357	2/420	9.05%
TBar (FL)	43/395	8/420	8.98%
Recoder (FL)	53/395	19/420	8.89%
RewardRepair (FL)	29/120	24/257	14.83%
TransplantFix (FL)	30/395	14/444	**4.44%**
RewardRepair (Non-FL)	45/120	45/257	20.0%
TransplantFix (Non-FL)	36/395	25/444	**3.48%**

x/y represents the number of correctly fixed bugs (i.e., "x") and the number of used bugs of the dataset (i.e., "y"). "(FL)" means the evaluation was performed in FL scenario and "(Non-FL)" means that in Non-FL scenario.

[Comparison in terms of Generalizability.] Table 3 presents the *DegradationRatio* score of five APR tools evaluated on both Defects4J v1.2 and v2.0 datasets. In FL scenario, there are five APR tools evaluated on both datasets. *TransplantFix achieves the best performance by reaching the lowest DegradationRatio, as compared to the other four APR tools.* Note that RewardRepair fixes the largest number of bugs in 257 bugs in Defect4J v2.0, as compared to all the five APR techniques. However, it fixes 29 bugs in 120 bugs in Defects4J v1.2, but this number decreases to 24 when twice more bugs (i.e., 257) are provided in Defects4J v2.0. Therefore, its *DegradationRatio* is the highest among the five APR tools. In Non-FL scenario, *TransplantFix* also achieves the best performance with the lowest *DegradationRatio* score (i.e., 3.48%) as compared to RewardRepair evaluated on both datasets.

One explanation for the better performance of *TransplantFix* over other APR tools is that *TransplantFix* does not rely on any manually crafted or automatically learned fix actions. Such crafted or learned fix actions might overfit a specific dataset. Instead, it extracts semantic fix actions from the donor method within the buggy project without any human intervention or adjustment of neural models, and thus is more generalizable than investigated APR tools.

4.5 Case Studies

While Chart-6 illustrated in Section 1 and 2 has shown the uniqueness of *TransplantFix*, we further performed two case studies on bugs in Defects4J to show how *TransplantFix* leverages its namespace transfer (i.e., the case study for Chart-17) and graph differencing (i.e., the case study for Chart-1) to produce correct patches for real-world bugs.

```
@@ -856,5 +856,5 @@
  public Object clone() throws CloneNotSupportedException {
-   Object clone = createCopy(0, getItemCount() - 1);
+   TimeSeries clone = (TimeSeries) super.clone();
+   clone.data = (List) ObjectUtilities.deepClone(this.data);
    return clone;
  }
```

(a) The human-written patch in TimeSeries class.

```
public Object clone() throws CloneNotSupportedException {
  XYSeries clone=(XYSeries)super.clone();
  clone.data=(List)ObjectUtilities.deepClone(this.data);
  return clone;
}
```

(b) A donor method in XYSeries class.

Figure 8: The case study for Chart-17.

Figure 8(a) presents a bug named Chart-17. Although the size of the human-written patch is relatively small, i.e., consisting of a single-line deletion and two-line insertion, this bug has never been fixed by other APR tools in both FL and Non-FL scenarios. Fixing this bug requires an APR technique to well address the insertions of multiple method calls, types, as well as variables. To fix this bug, *TransplantFix* first located the donor method shown in Figure 8(b) via the inheritance-hierarchy-aware code search, i.e., the `Clone()` method in `XYSeries` class that is inherited from the same parent class `Series` as the buggy class `TimeSeries`. Then, it leverages namespace transfer to construct the type mapping $\{XYSeries \rightarrow TimeSeries\}$, and finally apply the extracted fix actions (i.e., deleting the faulty statement and inserting the two donor statements) to produce a correct fix.

Figure 9(a) illustrates another bug named Chart-1, which corresponds to an if condition switch. Unlike existing APR techniques that fix the bug, *TransplantFix* fixes this bug in an interesting way, without relying on any manually crafted or automatically learned fix actions. Note that there is no donor code (i.e., "==" operator) in the donor method shown in Figure 9(b), *TransplantFix* identifies the semantic differences via our graph differencing based approach. That is, in the control flow graph representation, the buggy method shows the opposite behavior (i.e., `return result` when the condition `dataset != null` is true) to the donor method (i.e., `return result` when the condition `dataset != null` is false). Thus, *TransplantFix* identifies this semantic fix action (i.e., switching if condition) and correctly fixes it by applying the action.

```
public LegendItemCollection getLegendItems(){
  ...
  if (dataset != null) { // human fix: dataset == null
    return result;
  }
  if (...) {
    ... // Block A: 8 lines
  }else {
    ... // Block B: 8 lines
  }
  return result;
}
```

(a) The buggy method of Chart-1.

```
public LegendItemCollection getLegendItems(){
  ...
  if (dataset != null) {
    ... // Block A: same as the Block A in buggy method
  }
  return result;
}
```

(b) A donor method in a relative class of the buggy class.

Figure 9: The case study for Chart-1.

5 DISCUSSION

Potential of *TransplantFix* in complicated patch synthesis. We observed that there are about 20% of six-line-insertion patches or larger patches among the 36 patches generated by *TransplantFix* on Defects4J v1.2. Particularly, *TransplantFix* successfully synthesizes a patch of 15-line insertions that spans five chunks for Time-3 bug. It also synthesizes patches of eight-line insertions at a single hunk for both Math-13 and Chart-6. The three bugs all lie in the set of newly fixed bugs by *TransplantFix*. This indicates the potential of *TransplantFix* in synthesizing complicated patches for real-world bugs. When evaluated on Defects4J v2.0, this proportion has decreased into 8%. One explanation is that the new projects added in Defects4J v2.0 are much smaller, which makes it difficult to generate complicated patches. For example, the Cli project consists of 4 KLoc of source code, where the patches generated by *TransplantFix* for three Cli bugs are all one-line insertion patches. In such scenario, *TransplantFix* may need to search donor method from external projects. We leave the extension of code search as future work for exploring the full potential of *TransplantFix* in synthesizing more complicated patches.

Explainability of *TransplantFix*. Explainability of an APR technique (i.e., interpreting how the patch is generated) is important for greater adoption of APR techniques in the practical pipelines [19]. Recently, Noller et al. [52] reported that side products (e.g., identified faulty or fix locations) of APR techniques could enhance trust in APR-generated patches. In this work, the whole repair pipeline of *TransplantFix* is transparent and deterministic. Apart from the faulty or fix locations, *TransplantFix* could further provide more side products, including the identified donor method with similar functionalities as a repair reference, the edit graph that indicates the semantic differences between the donor and buggy method, and the exact fix actions applied to the buggy method.

These information provided by *TransplantFix* could benefit developers in patch correctness assessment, and thus enhance the developer trust on the patches generated by *TransplantFix*.

Next steps of *TransplantFix*. In this paper, we propose *TransplantFix* and present its promising results in terms of newly fixed bugs and generalizability. As future work, we envision two scenarios where *TransplantFix* could further fulfill its potential: 1) industrial deployment where *TransplantFix* could be integrated into the realistic repair pipeline of Repairnator [50] to aid manual debugging activities in the real world. 2) global-search based repair where *TransplantFix* extends the fix ingredient search scope to external projects. Accordingly, a set of new challenges arise as the inheritance relations may not work in this scenario and the namespace differences may be more significant.

Threats to validity. One threat to internal validity is that our literature review performed in Section 4.3 may fail to cover all existing APR techniques evaluated in Defects4J. To mitigate the threat, we manually checked all publications that cited the Defects4J publication [24] and identified qualified publications based on the inclusion criteria defined in Section 4.3. Another threat is that we extracted repair results of the 42 investigated APR tools from the literature, following prior practices [22, 37, 42, 78, 82]. However, such results are often derived from different hardware and timeouts, which may lead to a biased comparison. Considering the huge engineering effort and time cost required for re-running all APR tools with the same hardware and time budget (e.g., the RepairThemAll experiment [15]), we leave such experiment as future work.

A threat to external validity is that the bugs used for evaluating *TransplantFix* may not represent all real-world bugs. To mitigate the threat, we use a total of 839 real-world bugs from the most commonly used dataset Defects4J and its extended version. As future work, we plan to evaluate *TransplantFix* on more real-world datasets (e.g., Bears [43], Bugs.jar [54]).

6 RELATED WORK

6.1 Automated Program Repair

Automated program repair has attracted much attention in recent years. Since the proposal of the pioneer technique GenProg [66], a broad spectrum of APR techniques have been proposed. Goues et al. [19] divided existing APR techniques into three categories, i.e., heuristic search-based repair [22, 38, 46, 71, 79], constraint-based repair [16, 75], and learning-based repair [23, 25, 33, 78, 82]. *TransplantFix* lies in the scope of heuristic search-based repair that constructs and iterates the search space of program modifications.

A closely related set of work include SimFix [22] and ssFix [71]. Both APR techniques search donor code at the method snippet level. SimFix searches donor code within the buggy project, while ssFix leverages the Apache Lucene search engine to collect similar code snippets from a codebase. In terms of fix action extraction, both techniques perform AST differencing to extract fix actions from the donor code snippet, but SimFix further leverages mined fix actions to shape the search space. For donor code transfer, SimFix focuses on transfering variables. We notice that ssFix considers variables, types and methods. But it transforms all these entities into symbolized identifiers and builds all mappings at the same time, and it cannot deal with the donor transfer case when the needed entities (e.g., getShape() in Chart-6) do not appear in the buggy method. Another related work is LSRepair [38] that performs method-level search for method transplantation. LSRepair straightforwardly replaces the buggy method with the donor method when their method signatures are the same, or otherwise modifies four statement types (e.g., if statement). As compared to all these techniques, *TransplantFix* characterizes itself by an inheritance-ware method level approach within the buggy project, a graph based differencing approach to distill semantic fix actions, and a comprehensive donor method transfer approach.

6.2 Automated Software Transplantation

Barr et al. [8] proposed the notion of automated software transplantation (aka autotransplantation) that aims to automatically extract a functionality of the donor program and apply it into a host program. This motivated a series of avenues in the software engineering community. As reported by Marginean [44], existing autotransplantation approaches have been used in various areas, including automated program repair [60], security [77], testing [81], and functionality transplantation [59]. *TransplantFix* lies in the scope of using transplantation to automatically fix buggy programs.

Marginean identified two transplantation-based APR techniques via a literature review [44], i.e., GenProg [66] and CodePhage [60]. GenProg transfers donor code within the buggy program to the buggy location via genetic programming, and CodePhage operates on the binary code and transfers the correct code in one application into another one to fix errors (e.g., buffer overflow). Besides, we noticed that LSRepair [38] also mentions the notion of transplantation, and recently, Shariffdeen et al. [58] proposed an AST differencing based transplantation approach to transfer patches of vulnerabilities from one project to the buggy application. Different from these techniques, we present a new attempt to the APR community by exploring graph differencing based transplantation at the method level to mine semantic fix actions within the buggy project.

7 CONCLUSION AND FUTURE WORK

In this paper, We propose an inheritance-hierarchy-aware donor search approach to locate fix ingredients, a graph differencing based approach to distill semantic fix actions, and a comprehensive namespace transfer approach to adapt donor code. We highlight the uniqueness of *TransplantFix* via an extensive comparison involving 42 APR tools and an evaluation on 839 bugs in Defects4J v1.2 and v2.0. *TransplantFix* achieves the best performance in terms of both the number of newly fixed bugs and generalizability as compared to baseline APR tools. Also, it has shown its capability of synthesizing complicated patches consisting of at most eight-line insertions. In future work, we plan to extend *TransplantFix* with more scenarios (e.g., the integration into Repairnator) and experiments (e.g., evaluation with more datasets) as described in Section 5.

ACKNOWLEDGEMENT

This work is supported by the National Natural Science Foundation of China (No. 61872445).

REFERENCES

[1] [n. d.]. Artifact page of our study. [Online]. Available: https://github.com/DehengYang/TransplantFix, 2022.

[2] [n. d.]. The Java Language Specification. https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.4.2. last accessed: Jan. 2022.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[4] Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*.

[5] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. 2019. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* (2019).

[6] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[7] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[8] Earl T Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 257–269.

[9] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 378–381.

[10] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[11] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 637–647.

[12] Liushan Chen, Yu Pei, and Carlo Alberto Furia. 2020. Contract-based program repair without the contracts: An extended study. *IEEE Transactions on Software Engineering* (2020).

[13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).

[14] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 349–358.

[15] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 302–313.

[16] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*. 85–91.

[17] Gregory Gay and René Just. 2020. Defects4J as a Challenge Case for the Search-Based Software Engineering Community. In *International Symposium on Search Based Software Engineering*. Springer, 255–261.

[18] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[20] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th international conference on software engineering*. 12–23.

[21] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.

[22] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.

[23] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[24] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: 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*. 437–440.

[25] Sungmin Kang and Shin Yoo. 2022. GLAD: Neural Predicate Synthesis to Repair Omission Faults. *ISSTA* (2022).

[26] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. 2021. Evaluating Automatic Program Repair Capabilities to Repair API Misuses. *IEEE Transactions on Software Engineering* (2021).

[27] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 802–811.

[28] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *Empirical Software Engineering* 24, 6 (2019), 4071–4106.

[29] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1–45.

[30] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 314–325.

[31] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[32] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[33] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *2020 ACM/IEEE 42th International Conference on Software Engineering*. IEEE, 602–614.

[34] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *ICSE*.

[35] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 102–113.

[36] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[37] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[38] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F Bissyandé. 2018. LSRepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 658–662.

[39] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[40] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé François D Assise Bissyande, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *42nd ACM/IEEE International Conference on Software Engineering (ICSE)*.

[41] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 118–129.

[42] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–114.

[43] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[44] Alexandru Marginean. 2021. *Automated Software Transplantation*. Ph.D. Dissertation. UCL (University College London).

[45] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[46] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.

[47] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *International Symposium on Search Based Software Engineering*. Springer, 65–86.

[48] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions

of program repair approaches. In *Companion Proceedings of the 36th international conference on software engineering*. 492–495.
[49] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.
[50] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator patches programs automatically. *Ubiquity* 2019, July (2019), 1–12.
[51] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 180–190.
[52] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.
[53] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. 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*. 24–36.
[54] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 10–13.
[55] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.
[56] Ripon K Saha, Hiroaki Yoshida, Mukul R Prasad, Susumu Tokumoto, Kuniharu Takayama, and Isao Nanba. 2018. Elixir: an automated repair tool for Java programs. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 77–80.
[57] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 13–24.
[58] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2020. Automated Patch Transplantation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2020), 1–36.
[59] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 95–105.
[60] Stelios Sidiroglou-Douskos, Eric Lahtinen, and Martin Rinard. 2014. Automatic error elimination by multi-application code transfer. (2014).
[61] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.
[62] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in android apps. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 187–198.
[63] Ewan Tempero, James Noble, and Hayden Melton. 2008. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *European Conference on Object-Oriented Programming*. Springer, 667–691.
[64] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to design a program repair bot? insights from the repairnator project. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 95–104.
[65] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[66] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 364–374.
[67] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1–11.
[68] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 479–490.
[69] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.
[70] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–366.
[71] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.
[72] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.
[73] Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo Alberto Furia. 2020. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering* (2020).
[74] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 512–523.
[75] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.
[76] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F Bissyandé. 2021. Where were the repair ingredients for Defects4j bugs? *Empirical Software Engineering* 26, 6 (2021), 1–33.
[77] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 288–302.
[78] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. (2022).
[79] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* (2018).
[80] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 1 (2020), 1–53.
[81] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 665–676.
[82] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.