Is the Ground Truth Really Accurate? Dataset Purification for Automated Program Repair

Deheng Yang[†], Yan Lei[‡*], Xiaoguang Mao[†], David Lo[§], Huan Xie[‡], and Meng Yan[‡]
[†]National University of Defense Technology, Changsha, China, {yangdeheng13, xgmao}@nudt.edu.cn
[‡]Chongqing University, Chongqing, China, {yanlei, xiehuan.cs, mengy}@cqu.edu.cn
[§]Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg

***Abstract*—Datasets of real-world bugs shipped with human-written patches are intensively used in the evaluation of existing automated program repair (APR) techniques, wherein the human-written patches always serve as the ground truth, for manual or automated assessment approaches, to evaluate the correctness of test-suite adequate patches. An inaccurate human-written patch tangled with other code changes will pose threats to the reliability of the assessment results. Therefore, the construction of such datasets always requires much manual effort on isolating real bug fixes from bug fixing commits. However, the manual work is time-consuming and prone to mistakes, and little has been known on whether the ground truth in such datasets is really accurate.**

In this paper, we propose DEPTEST, an automated DatasEt Purification technique from the perspective of triggering Tests. Leveraging coverage analysis and delta debugging, DEPTEST can automatically identify and filter out the code changes irrelevant to the bug exposed by triggering tests. To measure the strength of DEPTEST, we run it on the most extensively used dataset (i.e., Defects4J) that claims to already exclude all irrelevant code changes for each bug fix via manual purification. Our experiment indicates that even in a dataset where the bug fix is claimed to be well isolated, 41.01% of human-written patches can be further reduced by 4.3 lines on average, with the largest reduction reaching up to 53 lines. This indicates its great potential in assisting in the construction of datasets of accurate bug fixes. Furthermore, based on the purified patches, we re-dissect Defects4J and systematically revisit the APR of multi-chunk bugs to provide insights for future research targeting such bugs.

***Index Terms*—bug dataset, automated program repair, dataset purification**

I. INTRODUCTION

Automated program repair (APR, hereafter) aims to automatically fix software bugs based on the specification (e.g., a formal specification [1]–[3] or a test suite [4]–[14]) indicating the expected behavior with no intervention of human developers [15]–[17]. The test-suite based APR techniques account for the most popular category [18], and have shown promise in fixing real-world bugs of diverse datasets [19]–[22]. For each buggy program in such datasets, a test suite containing at least one bug triggering test and a human-written patch are available for evaluating APR techniques. During the evaluation, the human-written patch is often considered as the ground truth by manual [23], [24] or automated [25]–[30] patch correctness assessment after the test-suite adequate patches are generated by APR techniques. Inaccurate ground truth may introduce noise and bias into the assessment activities, and thus may further negatively impact the reliability of assessment results [31], [32]. Therefore, for each bug of datasets used in APR, the associated human-written patch is expected to exclude the irrelevant code changes to the bug that is exposed by the triggering tests.

Unfortunately, the collection of accurate human-written patches is a challenging task. Mills et al. [33] reported that 63% of code changes are tangled in the bug fixing commits and irrelevant to the real bug fix. Similarly, Herbold et al. [31] showed that fewer than 40% of code changes contribute to the bug fix. Therefore, manual purification on isolating bug fixes from bug fixing commits is mandatory [31], [33]. However, such manual validation requires detailed project knowledge as well as the understanding of the intention of all code changes in the bug fixing commit. The lack of such knowledge of the dataset constructors as project outsiders may result in inaccurate isolation of human-written patches [32].

TABLE I: The construction of the three most popular datasets of Java real-world bugs used in APR.

	Collect bug fixing commits	Verify bug fixing commits	Obtain the minimal bug fix
Defects4J [19]	automatic	automatic	manual
Bugs.jar [21]	automatic	automatic manual	not done
Bears [20]	automatic	automatic manual	not done

It is worth noting that only one dataset among the three most popular datasets [18] of Java read-world bugs used in APR has conducted manual purification (i.e., obtaining the minimal bug fix) during construction. As shown in TABLE I, we notice that, during the construction process, the three datasets have all collected bug fixing commits automatically. They then verified if the bug fixing commit is indeed a commit for bug fixing by an automatic approach (i.e., Defects4J [19]) or combining both the automatic and manual approaches (i.e., Bugs.jar [21] and Bears [20]). However, after the verification is finished, the manual purification on the verified bug fixing commit, which may be tangled with unrelated code changes (e.g., features or refactorings) [19], [31], is not done by Bugs.jar and Bears. Our observation is consistent with that of Herbold et al. [31].

Motivated by the lack of manual purification and the absence of automated approaches assisting in dataset purification,

* Yan Lei is the corresponding author.

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00018

we propose an automated **D**atas**E**t **P**urification technique from the perspective of bug triggering **Test**s (DEPTEST in short). The goal of DEPTEST is to automatically obtain the purified human-written patch through the dynamic analysis on the buggy program and its test suite. While the real ground truth (i.e., the minimal human-written patch for a bug) is unknown and often difficult for developers to identify [31], [32], DEPTEST provides an automatic solution to approach the accurate ground truth by adequately taking advantage of the available artifacts (i.e., the test-suite containing bug triggering tests and the buggy program itself). DEPTEST first leverages the coverage information on both buggy and fixed version to identify the code changes not covered by bug triggering tests. Then, DEPTEST employs delta debugging to further infer the minimal human-written patch. The output of DEPTEST (i.e., the purified patch) is a subset of the human-written patch. To measure the strength of DEPTEST, we run DEPTEST on Defects4J, which is the most commonly used dataset in Java APR techniques and claims that all human-written patches it collected are free of unrelated code changes [19], [34], to explore if these patches can be further purified. Based on the purified patches, we re-dissect Defects4J in a fully automatic manner comparing against the dissection of Sobreira et al. [35] that combined manual and automatic approaches. Furthermore, we perform a systematic exploration on the APR techniques targeting multi-chunk bugs and provide insights for future research related to multi-chunk bug fixing.

The main contributions of our work include:

- We are the first to propose an automated dataset purification technique to identify and filter out the tangled code changes in human-written patches of datasets used in the context of APR. DEPTEST has shown its great potential in assisting in the construction of such datasets, by reducing 41.01% of human-written patches in Defects4J by 4.3 lines on average, even when Defects4J has already performed manual purification.
- We reveal the potential inaccuracy of the ground truth patches in Defects4J via purification. Based on the purified data, we re-dissect Defects4J to provide a more accurate anatomy of the dataset automatically.
- We identify the "pseudo multi-chunk human-written patches" in Defects4J, and perform a systematic revisit on the multi-chunk bug fixing of existing APR techniques, which provides insights for multi-chunk bug fixing. We also make all relevant artifacts of our study publicly available [36].

The remainder of the paper is structured as follows. Sec II illustrates two motivating examples. Sec III presents our DEPTEST approach, and Sec IV elaborates our study design. In Sec V we perform the data analysis, and in Sec VI we discuss. Finally, we present related work in Sec VII and conclude in Sec VIII.

II. MOTIVATING EXAMPLE

Defects4J is a dataset of real-world bugs targeting Java [19], where the human-written patch is claimed to be free of unrelated code changes via manual minimization process [34], and it is the most extensively used dataset in the evaluation of APR techniques. To investigate its popularity in APR, we perform a literature review on the APR techniques that have been evaluated on Defects4J. As shown in Fig. 1, we collect 47 APR techniques in total via manual validation from the 531 publications citing the literature of Defects4J [19], and mark the most effective APR technique each year in terms of number of correctly fixed bugs in Defects4J. During the process, we further observe some cases that expose the inaccuracy of some Defects4J human-written patches. For example, Fig. 2(a) presents the human-written patch of Math 65, consisting of two chunks. It is worth noting that the first chunk is actually a code refactoring, as indicated by its source code [37]. That is, human developers used *getChiSquare()* method to replace the deleted code in the first chunk. The real bug of Math 65 lies in the second chunk, where the "/" in line 258 of the buggy file should be replaced by "*". For this buggy chunk, CapGen [38] generated a test-suite adequate patch that is exactly identical to the second chunk of human-written patch, as presented in Fig 2(b). However, when researchers [39] evaluated the patch correctness of this patch, they labelled it as an incorrect patch, as this patch did not "fix" the first chunk (see Fig. 2(c)).

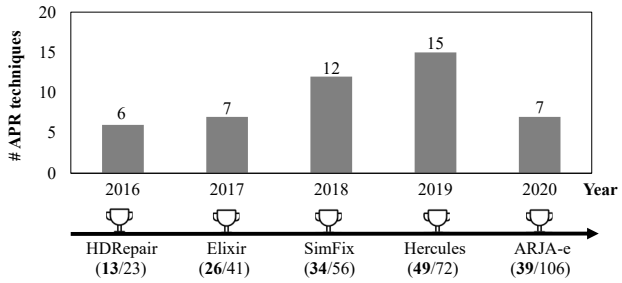

Fig. 1: The number of APR techniques evaluated on Defects4J. APR techniques performing best in each year in terms of number of correctly fixed bugs has been listed, in which APR techniques evaluated under perfect fault localization (e.g., CoCoNut [13]) are not considered. (X/Y) denotes the number of correctly/plausibly fixed bugs.

Preliminary observation 1: Non-buggy code changes (i.e., features or refactorings) in existing datasets of bugs that are used in automated program repair may pose threats to the reliability of manual patch correctness assessment. Furthermore, they pose threats to the correct anatomy of existing datasets for program repair. For example, Sobreira et al. [35] classified Math 65 as a two-chunk bug corresponding to 7 repair actions. However, when Math 65 is purified by eliminating the irrelevant code changes, it becomes a single-chunk bug corresponding to only 1 repair action.

Another case is the Time 2 of Defects4J. As shown in Fig. 3, the human-written patch of Time 2 consists of three chunks. To verify if this patch is accurate, we perform manual validation and a preliminary automatic coverage analysis. In the manual validation, we tend to agree with the Defects4J constructors that this human-written patch does not include

```
--- AbstractLeastSquaresOptimizer.java
+++ AbstractLeastSquaresOptimizer.java
@@ -239,10 +239,5 @@
     public double getRMS() {
-        double criterion = 0;
-        for (int i = 0; i < rows; ++i) {
-            final double residual = residuals[i];
-            criterion += residual * residual * residualsWeights[i];
-        }
-        return Math.sqrt(criterion / rows);
+        return Math.sqrt(getChiSquare() / rows);
     }

     /**
@@ -255,7 +250,7 @@
         double chiSquare = 0;
         for (int i = 0; i < rows; ++i) {
             final double residual = residuals[i];
-            chiSquare += residual * residual / residualsWeights[i];
+            chiSquare += residual * residual * residualsWeights[i];
         }
         return chiSquare;
     }
```

(a) The human-written patch provided by Defects4J for Math 65.

```
--- AbstractLeastSquaresOptimizer.java
+++ AbstractLeastSquaresOptimizer.java
@@ -255,7 +255,7 @@
         double chiSquare = 0;
         for (int i = 0; i < rows; ++i) {
             final double residual = residuals[i];
-            chiSquare += residual * residual / residualsWeights[i];
+            chiSquare += ((residualsWeights[i]) * residual) * residual
    ↪ ;
         }
         return chiSquare;
     }
```

(b) The test-suite adequate patch generated by CapGen for Math_65.

```
Human patch fixes two locations.
CapGen patch is equivalent to the second hunk of Human patch, but the
    ↪ first hunk is missing.
Hence CapGen patch is partial, and incorrect.
```

(c) The manual patch correctness assessment [40] by a research group [39] for CapGen patch of Math 65.

Fig. 2: A motivating example of the fact that irrelevant code changes tangled in the human-written patch lead to a wrong conclusion of manual patch correctness assessment.

unrelated code changes, as all three chunks are interdependent. However, when we run coverage analysis on both the buggy and fixed version of Time 2, we find that the third chunk is not covered by any bug triggering test. This indicates that the third chunk may be a tangled code change irrelevant to the bug exposed by failing tests. To verify this, we further check its commit message according to the commit id of Time 2 provided by Defects4J. We find that this bug fixing commit actually contains two bug fixes that correspond to two different bugs, while only one bug fix is exposed by the bug triggering test. In the context of automated program repair, the goal of test-suite based APR techniques is to fix the bug exposed by failing tests, and the bug cannot be exposed by any failing test is out of the scope of test-suite based repair. However, when using the human-written patches containing the unexposed bug fixes to assess the correctness of patches generated by test-suite based APR techniques, even if the APR produced identical bug fix to the exposed bug fix, it will be assessed as an incomplete fix according to existing criterion [41].

Preliminary observation 2: The bug fixes not triggered by any failing test but tangled in human-written patches are out of the scope of test-suite based APR. Such bug fixes are supposed to be excluded before performing patch correctness assessment.

Based on the two preliminary observations from two cases, we find that some human-written patches in Defects4J may not be accurate, and the limitation of manual validation on bug fix isolation does exist. That is, manual validation sometimes cannot find the unrelated code changes while automated analysis can do. Motivated by these issues in the context of automated program repair, we propose DEPTEST, an automated dataset purification technique, to help identify

```
--- org/joda/time/field/UnsupportedDurationField.java
+++ org/joda/time/field/UnsupportedDurationField.java
@@ -226,4 +226,7 @@
     public int compareTo(DurationField durationField) {
+        if (durationField.isSupported()) {
+            return 1;
+        }
         return 0;
     }

--- org/joda/time/Partial.java
+++ org/joda/time/Partial.java
@@ -216,4 +216,4 @@
     if (i > 0) {
         int compare = lastUnitField.compareTo(loopUnitField);
-        if (compare < 0 || (compare != 0 && loopUnitField.isSupported
    ↪ () == false)) {
+        if (compare < 0) {
@@ -448,4 +448,7 @@
         } else if (compare == 0) {
+            if (fieldType.getRangeDurationType() == null) {
+                break;
+            }
```

Commit message:
Fix NPE in Partial.with()
Also ensure unsupported duration fields are compared properly

Fig. 3: The human-written patch of Time 2.

and filter out the irrelevant code changes to the bug that is exposed by triggering tests.

III. APPROACH

Given a buggy program P_b of a dataset that can be used for evaluating test-suite based APR techniques, its fixed version P_f and human-written patch δ_h as well as a test suite T_s containing at least one bug triggering test T_t are also provided by the dataset. We formally present definitions related to dataset purification as follows:

Defenition 1 (code change). *A **code change** δ is a non-empty set: $\{\langle op, line\rangle \,|op \in \{del, ins\}, \forall op = ins, line \in P_b\}$ that can be applied to P_b.*

Defenition 2 (*apply*). *The function **apply**: $\{P_b, \delta\} \rightarrow P_v$ applies a code change δ into the buggy program to obtain a program variant P_v.*

A code change δ is a non-empty set containing pairs of operations and lines. The operation on each line can be a deletion or addition, indicating whether it deletes an existing line or adds a new line. The function *apply* is to apply any code change δ into the buggy program P_b to obtain a program variant.

Defenition 3 (*test*). *The function **test**: $(P_t, T_s) \rightarrow \{\surd, \chi\}$, where P_t is P_b or P_v.*

The function *test* takes a program P_t and the test suite T_s as input, and then run all test cases on the P_t. *test* finally outputs the result (i.e., pass or fail) of test execution.

Defenition 4 (human-written patch). *The **human-written patch** is a code change δ_h that satisfies the following constraints:*

$$\begin{cases} apply(P_b, \delta_h) = P_f \\ test(P_f, T_s) = \surd \end{cases} \tag{1}$$

Defenition 5 (purified patch). *A **purified patch** is a code change δ_p that satisfies the following constraints:*

$$\begin{cases} test(apply(P_b, \delta_p), T_s) = \surd \\ \delta_p \subseteq \delta_h \\ \forall \delta_i \subsetneqq \delta_p, test(apply(P_b, \delta_i), T_s) = \chi \end{cases} \tag{2}$$

In Definition 4, a code change is the human-written patch if it satisfies the constraints in Equation 1. That is, the buggy program after applying the code change is identical to the fixed program and can pass all test cases in the test suite. Furthermore, we define a purified patch δ_p as a code change that satisfies the Equation 2. This equation ensures that the purified patch is the minimal human-written patch. Based on these definitions, we can further give an axiom as follows:

Axiom 1 (An inaccurate human-writtem patch).

$$\delta_h \textit{ is inaccurate } \iff \delta_h \neq \delta_p$$

To identify if a human-written patch δ_h is accurate, we need to first obtain the purified patch δ_p, based on which we can say that the human-written patch is inaccurate and can be further *purified* when it is not identical to the purified patch δ_p that corresponds to the minimal δ_h.

Based on the above definitions and axiom, our work of identifying whether the ground truth is really accurate in datasets that can be used in APR has been formulated as the work of obtaining δ_p for each P_f. As illustrated in Sec I and Sec II, manual analysis requires detailed project knowledge and is prone to mistakes. Therefore, we propose an automated approach (i.e., DEPTEST) to obtain the purified patches from human-written patches.

Algorithm 1 The algorithm of DEPTEST.

Input: Buggy program P_b
Input: Test suite T_s and bug triggering tests T_t
Input: Human-written patch δ_h
Output: A purified patch δ_p

```
1: function MAIN
2:     P_f ← apply(P_b, δ_h)
3:     δ_cov ← coverageAnalysis(P_b, P_f, T_t, δ_h)
4:     δ_p ← deltaDebugging(P_b, T_s, delta_cov)
5:     return δ_p
6: end function

7: function COVERAGEANALYSIS(P_b, P_f, T_t, δ_h)
8:     δ_cov ← ∅
9:     cov_b ← ∅
10:    cov_f ← ∅
11:    for T'_t in T_t do
12:        cov_b ← cov_b ∪ getCovByExec(P_b, T'_t)
13:        cov_f ← cov_f ∪ getCovByExec(P_f, T'_t)
14:    end for
15:    for ⟨op, line⟩ ∈ δ_h do
16:        if op = del and line ∈ cov_b then
17:            δ_cov ← δ_cov ∪ {⟨op, line⟩}
18:        else if op = ins and line ∈ cov_f then
19:            δ_cov ← δ_cov ∪ {⟨op, line⟩}
20:        end if
21:    end for
22:    return δ_cov
23: end function

24: function DELTADEBUGGING(P_b, T_s, δ_cov)
25:    n ← 2
26:    repeat
27:        complementPass ← false
28:        subsets ← split(δ_cov)
29:        for subset ∈ subsets do
30:            complement = minus(δ_cov, subset)
31:            if test(apply(P_b, complement), T_s) = √ then
32:                δ_cov ← complement
33:                n ← Math.max(n − 1, 2)
34:                complementPass ← true
35:                break
36:            end if
37:        end for
38:        if complementPass = false then
39:            if n = |δ_cov| then
40:                break
41:            end if
42:            n ← Math.min(n ∗ 2, |δ_cov|)
43:        end if
44:    until |δ_cov| < 2
45:    return δ_cov
46: end function
```

As shown in Algorithm 1, motivated by the examples we observe and discuss in Sec II, DEPTEST employs coverage analysis and delta debugging to purify human-written patches

(lines 1 – 6). DEPTEST first obtains the fixed version by applying the human-written patch δ_h to the given buggy program (line 2). Then, it performs coverage analysis using bug triggering tests T_t (line 3). In coverage analysis (lines 7 – 23), DEPTEST collects the covered lines of each bug triggering test by executing them on both the buggy and fixed version (lines 11 – 14). Then, all the pairs of $\langle del/ins, line \rangle$ in δ_h not included in the set of lines covered by T_t on buggy/fixed program will be excluded (lines 15 – 21). Based on the output of *coverageAnalysis()*, DEPTEST further leverages delta debugging to obtain the minimal subset of $delta_{cov}$. Delta debugging [42], [43] is an automated technique to isolate and minimize the failure-inducing input or code changes. It is also employed in GenProg [4], [44] and ARJA-e [45] to minimize the code edits of candidate patches. In this work, DEPTEST uses delta debugging to further minimize the δ_{cov} for obtaining a minimal subset of δ_h (lines 24 – 46). The function *deltaDebugging()* we implemented is consistent with the algorithm demonstrated in the literature [42].

Note that the DEPTEST algorithm is performed at the granularity of code line, which can facilitate the coverage analysis and delta debugging. For example, the irrelevant elements not triggered by failing tests can be directly identified and according to the covered lines information from coverage analysis.

TABLE II: The mechanism of DEPTEST for obtaining the purified patch δ_p.

		Code changes	
		Buggy	Non-buggy
Exposed by bug triggering tests	Yes	δ_p	#1
	No	#2	#3

As presented in TABLE II, we classify the code changes of human-written patches into four types in terms of whether they are buggy or exposed by bug triggering tests. By performing coverage analysis, DEPTEST identifies and excludes the type #2 and #3 code changes. With delta debugging, DEPTEST can further exclude the type #1 code changes and obtain the purified patch (i.e., δ_p).

The major risk of using DEPTEST is whether any part of the real bug fix in the human-written patch δ_h will be filtered out (i.e., false positives) during the purification, as the specification used by DEPTEST is the test suite rather than a formal and complete specification of the buggy project. We discuss in detail the risk and soundness of DEPTEST in Sec VI based on our verification on the purification data.

IV. STUDY DESIGN

A. Research Questions

Our paper aims to answer the following three research questions:

RQ1: How effective is DEPTEST in purifying the dataset of real-world bugs that can be used in test-suite based APR?

While bug datasets widely used in the evaluation of Java test-suite based APR techniques perform manual purification (e.g., Defects4J [19]) even do not perform any purification (e.g., Bears [20] and Bugs.jar [21]) for minimizing the bug fixes, we propose an automated purification technique for assisting in the purification process. The goal of **RQ1** is to investigate if our automated purification technique DEPTEST is effective in identifying and filtering out the irrelevant code changes in human-written patches of the dataset, even when the dataset (i.e., Defects4J [19]) has already been manually purified.

RQ2: How are general properties of patches distributed in the dataset after purification by DEPTEST?

Based on the purified data on Defects4J, we re-dissect the Defects4J. Similar to the work of Sobreira et al. [35], we consider a series of quantitative and qualitative properties. Among these properties, we mainly focus on the repair patterns, which are an important source for bug fixing and are automatically extracted in our paper. While previous study [35] is based on the original dataset, **RQ2** aims to explore how general properties including repair actions and patterns are distributed in the dataset after purification by DEPTEST.

RQ3: What is the real challenge of fixing multi-chunk bugs?

As shown in Sec II, some multi-chunk bug fixes in Defects4J (e.g., Math 65 presented in Fig. 2) are actually single-chunk bug fixes. We call such multi-chunks bug fixes as "***pseudo multi-chunk fixes***". Previous studies [35] on such human-written fixes may be inaccurate to a certain extent. For example, the analysis result of Sobreira et al. [35] on Math 65 contains 2 chunks and 7 repair actions, while the real bug fix of Math 65 (i.e., the second chunk as shown in Fig 2(a)) consists of only 1 chunk and 1 repair action. Therefore, **RQ3** targets investigating how many such "*pseudo multi-chunk fixes*" exist in Defects4J, and what is the real challenge of automated repair on multi-chunk bugs.

B. Subject Dataset

To answer our research questions, we define the following inclusion criteria to select the subject dataset:

- **Criterion 1:** The bugs of the dataset must be collected from real-world projects. We focus on studying the dataset constructed from real world bug fixing commits, which is strongly recommended to perform bug fix isolation (i.e., purification) [31], [33]. Therefore, this criterion excludes QuixBugs [46] and IntroclassJava [47].
- **Criterion 2:** The dataset with high popularity in evaluation of test-suite based APR techniques will be prioritized. As our work aims to study if the ground truth patch in the dataset is really accurate for test-suite based APR techniques, we prioritize the dataset that is extensively used by such APR techniques for our experiment.

Based on the two criteria, we finally select Defects4J [19] as our subject dataset. Defects4J is the most popular dataset used in the evaluation of test-suite based APR techniques, which covers 47 APR techniques as we investigate in Sec II. Furthermore, comparing against Bugs.jar [19] and Bears [20], Defects4J performs manual purification during construction, as shown in TABLE I. As a result, we perform DEPTEST on Defect4J bug fixes to explore whether automated purification

approach can detect the irrelevant code changes which are omitted by manual purification. Concretely, we use the Defects4J v1.4.0 [19], which contains 395 bugs collected from 6 large open-source projects.

C. Experimental Setup

For the implementation of DEPTEST, we leverage the Jacoco [48], which is also used in GZoltar v1.7.2 [49] for fault localization, to perform the coverage analysis. For the model of delta debugging, we follow the delta debugging algorithm that is publicly available [50]. The implementation of DEPTEST is consistent with the Algorithm 1. In addition, our purification experiment is conducted on a 64-bit Ubuntu server with two Intel Xeon CPUs and 8GB RAM. For each purification on a given human-written patch, we set the time budget as 2 hours. The purification process terminates when the purified patch is obtained or the time budget expires.

V. RESULT ANALYSIS

A. RQ1: Effectiveness of DEPTEST

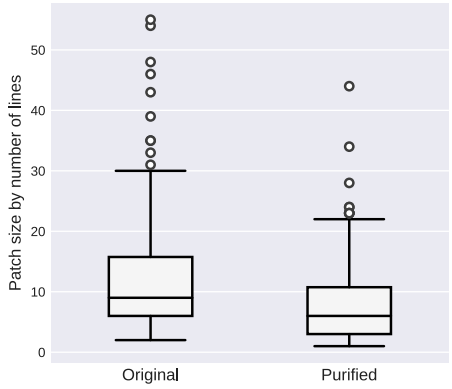

Fig. 4: The distribution of patch size by number of lines for original patches and purified patches (i.e., $|delta_h|$ and $|delta_p|$).

To evaluate the effectiveness of DEPTEST, we measure differences between the purified human-written patch δ_p produced by DEPTEST and the original human-written one δ_h in Defects4J. We first analyze the differences in terms of patch size by number of lines. 41.01% (i.e., 162/395) of human-written patches are identified by DEPTEST as *inaccurate* ground truth according to Axiom 1, where the $\delta_p \neq \delta_h$. For the 162 cases, we further analyze their purification sizes in terms of reduced code change lines. As shown in Fig. 4 that visualizes the distributions for all purified cases, the patch size by number of lines of original patches has been reduced obviously. As displayed in TABLE III, the human-written patches in Defects4J are reduced by 4.3 lines on average. The maximal purification occurs in Lang 25, reaching up to a reduction of 53 lines in total. We then calculate the average percentage of reduced lines by the following formula:

$$\textit{purification ratio} = \frac{\sum_{i=1}^{|\textit{purifiedSet}|} \frac{(|\delta_{hi}|-|\delta_{pi}|)}{|\delta_{hi}|}}{|\textit{purifiedSet}|} \times 100\%, \quad (3)$$
$$\textit{purifiedSet} = \{\langle \delta_{pi}, \delta_{hi} \rangle \, | \delta_{pi} \subsetneqq \delta_{hi}, i \in [1, 395]\}$$

As shown in Equation 3, the *purification ratio* measures the average percentage of the number of reduced lines for *purifySet* that contains 162 purified cases. Concretely, 35.02% (i.e., the purification ratio) of code change lines are reduced on average, which indicates a consideration improvement brought by DEPTEST. Furthermore, we find that the average time cost of purifying a bug by DEPTEST is 605.82 seconds, which indicates a relatively low overhead.

TABLE III: The purification size in terms of reduced lines and the result of statistical tests for evaluating the magnitude of purification.

Min	Max	Mean	*P*-value	A_{12}
1	53	4.3	1.1353e-28	0.6579

To qualitatively and quantitatively evaluate the improvement of DEPTEST over the original Defects4J dataset, we further perform statistical tests and compute effect size. Consider the paired data of original and purified patches and their heavy-tailed distributions shown in Fig. 4, we employ the non-parametric Wilcoxon signed-rank test to evaluate if the purification size is statistical significant. The null hypothesis is that two paired groups share the same distribution, and we reject the null hypothesis with $\alpha = 0.05$. Furthermore, we apply a non-parametric effect size measure called non-parametric Vargha and Delaney's A_{12} statistic [51], [52] to measure the magnitude of improvement, where A_{12} over 0.56, 0.64 and 0.71 represents a small, medium and large effect size respectively. As shown in TABLE III, DEPTEST achieves statistically significant improvement over the original human-written patches in Defects4J, as indicated by the *p-value*, and reaches a "medium" effect size according to the A_{12} statistic. The above result analysis indicates that DEPTEST is effective on purifying patches in Defects4J, even when Defects4J has already experienced manual purification.

For human-written patches that cannot be purified by DEPTEST, the main reason is that these human-written patches are minimal before collected by DefectsJ constructors, or become minimal after the manual purification of constructors. One representative fact is that there are 143 human-written patches consisting of less than or equal to 3 code change lines, of which only 13 can be purified.

To further illustrate the effectiveness of DEPTEST, We take a closer look at the purified patches and obtain some interesting findings. Take Chart 5 as an example, both the constructors [19] and researchers as end users [53], [54] of Chart 5 in Defects4J do not realize that the second chunk is a refactoring, even they performed manual purification or carefully analyzed this bug fix. In fact, when *this.allowDuplicateXValues* is true, the function will directly *return null* without executing the second chunk. When *this.allowDuplicateXValues*

is false, the function will execute the second chunk, but *!this.allowDuplicateXValues* in the second chunk becomes true and will never impact the value of the if expression. Therefore, the second chunk of the human-written patch is a refactoring to remove the redundant code. While the constructors or end users fail to identify such irrelevant change, DEPTEST successfully identifies and filters out it within 4 minutes. It fails to identify the second chunk, which is covered by failing tests, during coverage analysis. But it soon identifies the second chunk as a type #1 code change (see TABLE II) and obtains the purified patch via delta debugging.

Another instance is the purification of Time 2. The purified patch produced by DEPTEST only keeps the first chunk of the human-written patch, which originally consists of three chunks (see Fig. 3). DEPTEST first excludes the third chunk during coverage analysis, then it uses delta debugging to filter out the second chunk. The whole activity of DEPTEST on this bug is finished within 3 minutes. On the other hand, we authors tried to manually purify it, we excluded the third chunk in a very short time when we noticed the bug fixing commit message of Time 2. However, we devoted hours of attempts to understanding and debugging the Time 2 project, and finally found it is a refactoring (i.e., we find that the expression *(compare != 0 && loopUnitField.isSupported() == false)* can never be true after carefully checking its buggy classes).

Surprisingly, we do not find any false positive (i.e., the DEPTEST wrongly purifies the part of real bug fix) produced by DEPTEST. We discuss this issue in detail in Sec VI. We also study the contributions of coverage analysis and delta debugging in DEPTEST to the purification in terms of reduced code change lines. Among the 162 purified human-written patches, coverage analysis purified 58 lines in total for 23 human-written patches, costing 7.3 seconds on average. For delta debugging, it purified 635 lines in total for 155 human-written patches, of which the average time cost is 561.6 seconds. Coverage analysis is less effective but more efficient, while delta debugging performs more effectively but less efficiently. Both complement each other well in DEPTEST.

Answer to RQ1: Even in a dataset that has performed manual purification, our DEPTEST tool has purified 41.01% of Defects4J human-written patches, where 35% of code changes (i.e., 4.3 lines) of each patch are reduced on average. Such improvement corresponds to a statistical significance and a "medium" effect size. Furthermore, our preliminary comparison between the DEPTEST and manual purification indicates its great potential to effectively assist constructors or end users in identifying the minimal patch with very low runtime overhead.

B. RQ2: Re-dissection of Defects4J

As shown in Sec II, Defects4J is intensively used in the evaluation of test-suite based APR techniques. Due to the lack of formal specification of the buggy project, the human-written patch is always considered as the ground truth to assess if a test-suite adequate patch generated by APR techniques is correct. The inclusion of irrelevant code changes to the human-written patch will pose threats to the patch correctness assessment. For example, some researchers in APR community have already realized the inaccuracy of human-written patches as the ground truth. For example, authors of ACS [10] pointed out that the third chunk of the human-written patch of Math 93 [55] is not related to the bug, so they manually purified such human-written patch (i.e., exclude the third chunk) before performing patch correctness assessment. However, there might exist more such type of cases that are still not discovered. Therefore, we propose DEPTEST and perform a re-dissection on Defects4J to provide more accurate ground truth for evaluation of APR techniques.

TABLE IV: Seven descriptive statistics for properties of human-written patches before and after purification. X/Y denotes the value of the original/purified patch.

	Min	25%	50%	75%	90%	95%	Max
# inserted Lines	0/0	1/1	3/3	8/6	14/12	20/16	49/39
# deleted Lines	0/0	0/0	1/1	2/2	5/3	8/6	33/21
Patch size	1/1	2/2	5/4	10/8	19/15	24/21	55/44
# Chunks	1/1	1/1	2/2	3/3	5/5	8/7	20/20
# Files	1/1	1/1	1/1	1/1	1/1	2/2	7/7
# Classes	1/1	1/1	1/1	1/1	1/1	2/2	7/7
# repair actions	0/0	1/1	2/1	3/3	5/5	8/6	22/24
# repair patterns	0/0	7/6	11/11	21/19	33/30	40/36	70/70

In the re-dissection, we first analyze the quantitative and qualitative properties we collected for Defects4J. We leverage 7 descriptive statistics to summarize the distributions of the 8 key properties of Defects4J human-written patches, including both the original patch and the purified patch. As shown in TABLE IV, in terms of the patch size at the granularity of code line (i.e., the number of inserted, deleted, and patch size), the difference between original patches and purified ones is not obvious for the first half of patches. With the patch size growing, the difference is increasing, which indicates that the DEPTEST approach mainly purified the human-written patches consisting of more code lines. For example, the largest patch size of original human-written patches is 55 (i.e., Math 92). DEPTEST has purified this patch into a 19-line patch. Interestingly, for the second largest human-written patch Lang 25 consisting of 54 code change lines, DEPTEST purified it into a single-line patch. We further check its bug report site, where we find that the remained single-line bug fix is also found in the bug report, and other tangled code changes are not relevant to the triggering tests. With such purified information, end uses can gain more accurate information for evaluating test-suite based APR techniques.

In terms of the number of chunks and their spreading (i.e., number of modified files and classes), the difference between original patches and purified ones is minor. 90% of patches

TABLE V: Top-5 most frequently used repair patterns of patches for the purified Defects4J.

INS			DEL		
Source node	Target node	Freq. (%)	Source node	Target node	Freq. (%)
SName	MethodInv	9.9	SName	MethodInv	2.5
Operator	InfixExpr	6.1	Operator	InfixExpr	1.2
SName	InfixExpr	4.6	SName	QName	0.9
SName	QName	2.4	SName	InfixExpr	0.7
InfixExpr	IfStmt	2.3	MethodInv	InfixExpr	0.6
UPD			**MOV**		
Source node	Target node	Freq. (%)	Source node	Target node	Freq. (%)
SName	MethodInv	0.9	Operator	InfixExpr	0.5
VDF	VDStmt	0.5	ExprStmt	IfStmt	0.5
InfixExpr	IfStmt	0.4	SName	MethodInv	0.4
Operator	InfixExpr	0.2	SName	InfixExpr	0.3
MethodInv	ReturnStmt	0.2	IfStmt	IfStmt	0.2

SName: SimpleName; MethodInv: MethodInvocation
IfStmt: IfStatement; InfixExpr: InfixExpression
QName: QualifiedName; VDF: VariableDeclarationFragment
VDStmt: VariableDeclarationStatement
ExprStmt: ExpressionStatement; ReturnStmt: ReturnStatement

correspond to 6 chunks and 1 modified file and class. Also, both purified and original version share the maximal value of the three properties. We notice that the patch of Math 6 has 7 modified files and classes. In this case, even though DEPTEST has reduced the human-written patch of Math 6 largely, from 30 code change lines and 17 chunks to 12 lines and 9 chunks, the number of modified classes are still remained unchanged.

For the repair actions and patterns, the reduction of repair patterns brought by DEPTEST is larger than that of repair actions. We further observe from TABLE IV that while all the value of properties for purified patches are reduced, the maximal value of its repair actions slightly increases from 22 to 24. Note that repair actions are extracted from the GumTree output, such slight increase is reasonable as in some cases the repair action corresponding to a chunk will be split into several smaller repair actions.

To obtain a comprehensive view of how the abstract repair patterns are distributed in Defects4J patches, we collect in total 3146 different repair patterns and 48,493 occurrences of these patterns by GumTree from all 395 human-written patches of Defects4J after purification. Table V covers the four types of code changes provided by GumTree, including the insertion (INS), deletion (DEL), update action (UPD) and move action (MOV) of AST nodes. For each type, the top-5 most frequently applied repair patterns in human-written patches of the purified Defects4J are listed. As shown in the table, the INS and DEL action of AST nodes occur much more frequently than UPD and MOV. In addition, the SimpleName, MethodInvocation and InfixExpression commonly exist in each type of patterns, which indicates that the human-written bug fixes include many repair actions on the three types.

TABLE VI: The purification data of multi-chunk human-written patches in Defects4J.

Patch type	# Patches
Total	395
Multi-chunk	245
Multi-chunk purified by at least a line	142
Multi-chunk purified by at least a chunk	68
Multi-chunk purified into a single-chunk	28

Answer to RQ2: We re-dissect the Defects4J dataset based on our purification that excludes irrelevant code changes, to obtain a more accurate comprehension of the dataset. The complexity in terms of patch size and repair patterns is reduced obviously after purification. Furthermore, our analysis on repair patterns indicates the two defective AST nodes (i.e., MethodInvocation and InfixExpression) and the need to carefully collect the defect context containing SimpleName which is also very frequently applied in bug fixes. All our data of re-dissection are publicly available to facilitate a further exploration of end users [36].

C. RQ3: Revisit of Multi-chunk bug fixing

Automated program repair of multi-chunk bugs is a challenging task, with most of APR techniques limited to single-edit patch generation and only a few APR techniques explicitly claiming to support fixing such bugs. The tangled code changes in the multi-chunk human-written patches further increase the difficulty for the APR community to identify the real challenge of automated repair of such bugs. As shown in TABLE VI, our DEPTEST tool successfully purified 142 out of 245 (i.e., 57.96%) multi-chunk human-written patches with at least one-line reduction. Among the 142 purified cases, the multiple chunks of 68 cases are reduced by at least one chunk. Interestingly, in this process, we observed 28 multi-chunk human-written patches that are reduced into a single-chunk patch through purification. For instance, the human-written patch of Time 2 presented in Fig. 3 and analyzed in Sec V-A is purified by DEPTEST into a single-chunk patch. We call such type of patches "pseudo multi-chunk patches", which correspond to 28 patches in total as indicated by TABLE VI. Such patches that contain irrelevant code changes (e.g., refactorings or bug fixes not relevant to the bug) are misclassified as multi-chunk patches, and thus are supposed to be excluded before we revisit multi-chunk bug repair.

Based on our purification data of multi-chunk bugs, we systematically revisit multi-chunk bug repair to explore the real challenges of this area. First, we explore how many multi-chunk bugs existing APR techniques can correctly fix. We select APR techniques for revisit with the following inclusion criteria:

- The APR technique is evaluated on Defects4J dataset, as our purification experiment is based on Defects4J. In this step, we collect 47 APR techniques [36].
- The correct patches generated by APR techniques must be publicly available and correspond to at least one real multi-chunk bug (i.e., not one of the 28 identified "pseudo

TABLE VII: Multi-chunk bugs correctly fixed by APR techniques.

APR	Bug ID
HDRepair [58]	M-22
Nopol [7]	L-55
PraPR [59]	L-10
ssFix [60]	CL-115
JAID [61]	C-26, CL-40
ARJA [53]	L-20, 35, M-22, 98
kPAR [62]	C-4, CL-2, M-15, 89, T-7
LSRepair [63]	C-4, L-46, 48, M-89, Mo-13
GENPAT [64]	CL-115, L-47, 60, M-4, 22
AVATAR [65]	C-14, 19, CL-2, L-10, M-4, 46, T-7
SimFix [11]	CL-115, L-41, 50, 60, M-35, 71, 98, T-7
ACS [10]	C-14, 19, L-35, M-3, 4, 25, 35, 61, 89, 90, 93, 99
TBar [12]	C-4, 14, 19, CL-2, 21, L-10, 47, M-4, 22, 35, 77, 89, 98, T-7

M means Math, and L represents Lang. C and CL denote Chart and Closure respectively, and Mo is Mockito.

TABLE VIII: Bug types summarized from existing correctly fixed multi-chunk bugs.

Bug Type	Bug ID
Single-edit	C-4, 26, L-48, 55, CL-2, 40, M-3, 25, 89, T-7
Independent	C-14, 19, L-20, 35, 47, 60, M-4, 22, 35, 98, 99
Interdependent	CL-21, 115, L-10, 41, 46, 50, M-15, 61, 77, 90, Mo-13

multi-chunk bugs") in Defects4J. A publicly available patch is indispensible for our systematic revisit. As a result, a set of APR techniques (e.g., Hercules [56], ARJA-e [45], and ARJA-p [57]) are not included.

Based on the above criteria, we finally identified 13 APR techniques and 32 multi-chunk bugs that can be correctly fixed by at least one APR technique. We further classified these multi-chunk bugs into the following three categories based on the human-written patch after purification:

1) *Single-edit*: the human-written patch of this bug consists of only a single edit. For example, the patch of Chart 4 corresponds to an addition of an if statement, while its second chunk consists of only an "}".
2) *Independent*: the chunks of the human-written patch are independent of each other. Such bugs often have sufficient triggering tests to expose each buggy chunk. For example, the patch of Chart 14 consists of 4 addition chunks and each chunk is exposed by a failing test case.
3) *Interdependent*: the chunks of the human-written patch are interdependent with each other. Only modifying one chunk of the bug may generally lead to a failure. For instance, the human-written patch of Mockito 13 requires the modification on both modifying the if expression and inserting an else block at the other place.

[*Some multi-chunk bugs are as easy as single-chunk ones.*] As shown in TABLE VIII, 10 out of 32 correctly fixed bugs lie in the category of single-edit multi-chunk bugs. According to the previous definition of multi-chunk bugs [35], these bugs (e.g., the human patch of Math 3 that is separated by a comment) are classified as multi-chunk bugs. Essentially, they can be interpreted as single-chunk bugs.

[*Promising performance on independent multi-chunk bugs.*] We observe that existing multi-chunk APR techniques, which fully leverage the triggering test information, generally work well on the independent multi-chunk bugs. Such APR techniques include ACS, SimFix, and ARJA. For ACS [10], one of its templates is based on the oracle provided by triggering tests, which can facilitate the condition synthesis. For SimFix [11], the test case purification it used can help it purify a single failing test case into multiple failing cases, which can then be used for fixing the bug chunk by chunk (e.g., Math 98). Similarly, ARJA [53] leverages a test filtering strategy and multi-objective algorithm to fix multi-chunk bugs.

[*Limitation in scaling up to complex interdependency.*] Existing APR techniques are trying to understand the repair logic of interdependency in multi-chunk bugs, as indicated by TABLE VII and VIII where 10 interdependent bugs can be correctly fixed. However, the correct patches generated by APR techniques on this bug are most often a single-edit patch. In addition, most interdependent bugs listed in TABLE VIII consist of no more than 3 chunks. This is due to the common assumption of many APR techniques that only fixing one faulty location can fix the bug, which further prohibits these techniques to scale up to the complex interdependency.

Answer to RQ3: We systematically revisit existing APR techniques capable of fixing multi-chunk bugs based on our purification results that exclude "pseudo multi-chunk fixes". We further classify multi-chunk bugs into three categories and analyze both the strength and weakness of existing APR techniques in multi-chunk bug repair. We suggest that efforts on re-constructing the assumption of APR techniques are necessary for embracing automated fixing of more complex interdependent bugs.

VI. Discussion and Implications

DEPTest for dataset constructors. As presented in Sec V-A, DEPTest shows its promising performance with a statistical significance even in a dataset that has performed manual purification. Furthermore, in our implementation, DEPTest is enabled to provide detailed information of why the code change lines are purified by DEPTest (e.g., the lines are not covered by any failing test) rather than merely providing the purified patch. In this way, DEPTest can assist dataset constructors in identifying irrelevant code changes to the real bug fix by providing the information that constructors may omit or fail to find. DEPTest is expected to alleviate the burden of manual purification for constructors of datasets which can be used in APR, and improve the accuracy of human-written patches used as ground truth in APR.

DEPTest for end users. DEPTest provides more accurate human-written patches for 162 out 395 patches in Defects4J. The purification data including the purified patch is publicly available as a YAML file for each patch [36], which

can be a reference for end users who need to use human-written patches as the ground truth. For example, when end users assess the correctness of a patch generated by APR on Math 93, this patch only corresponds to the first two chunks of the original human-written patch in Defects4J. If the end users check our purified patch on Math 93 which purifies the third chunk, they then just need to focus on comparing the first two chunks with the APR patch during assessment. Otherwise, they have to manually verify if the third chunk is relevant to the real bug fix as Xiong et al. [10] did as discussed in Sec V-B, to obtain reliable assessment results.

DEPTEST for APR that goes beyond the test-suite. We notice that some APR techniques (e.g., Hercules [56]) can correctly fix the multi-chunk patches (e.g., Closure 4), of which some chunks cannot even be exposed by any failing test. From the perspective of test-suite based APR, such chunks cannot be localized and cannot be further fixed [62], and will also be filtered out by DEPTEST. For such techniques that go beyond the scope of traditional test-suite based APR, DEPTEST provides some explanations, from the perspective of test-suite, to the comparison results of such techniques against traditional test-suite based APR techniques (e.g., only Hercules correctly fixed the Closure 4 [56]).

Limitations of DEPTEST. As pointed out in III, there is a risk for DEPTEST that it might filter out the part of the real bug fix in the human-written patch δ_h during purification, due to the unavailability of the formal and complete specification of the project. To measure this risk, we manually verified the purified patches of DEPTEST. However, such verification requires not only detailed project knowledge but also much time cost, similar to manual purification [32]. Therefore, we randomly selected 81 (i.e., a half as a statistical size) out of the 162 purified cases without any bias for verification. To obtain reliable results, three authors in our group independently conducted the verification and discussed on inconsistent verification results until an agreement is reached. We find that among the 81 verified cases DEPTEST produced no such patches where part of real bug fixes are eliminated. An explanation is that DEPTEST focuses on filtering out the not covered (i.e., unrelated) statements during coverage analysis and the loosely coupled (i.e., loosely related [32]) code changes tangled in real bug fixes during delta debugging. As a result, the complete logic and components of the real bug fix are maintained during the purification.

Interestingly, we find that DEPTEST failed to obtain minimal patches in some cases. For example, it only reduced 1 line for Math 65, while the ideal reduction is 7 lines. The reason is that the delta debugging we implemented in DEPTEST does not consider the program syntax and might fail to obtain the minimal set in complex cases [66]. We leave the further improvement of delta debugging algorithm as our future work.

Threats to validity. Our implementation of DEPTEST may potentially contain bugs. To mitigate this threat, we double-check our code implementation and make all relevant artifacts publicly available. In addition, our study only considers Defects4J dataset, which may fail to represent other datasets. We mitigate this threat by carefully reviewing existing datasets and select a dataset (i.e., Defects4J) that is harder to be purified to measure the effectiveness of DEPTEST. Our future work will explore if DEPTEST can achieve a larger improvement on other datasets that do not perform manual purification.

VII. Related Work

Tangling code changes in bug fixing commits are common and may pose noise and bias to relevant venues of research [32], [33]. To decompose the tangling code changes, Barnett et al. [67] proposed an automated approach based on static analysis to assist developers in code review. Recently, Herbold et al. [31] conducted a large-scale manual validation to identify the tangled code changes in bug fixing commits through crowd working. Unlike their work, our DEPTEST is a fully automated technique that employs dynamic analysis (i.e., coverage analysis and delta debugging [42]) and targets assisting benchmark constructors in dataset purification and providing end users a more accurate human-written patch as the ground truth.

Automated program repair (APR) is a very active research area in software engineering, with a number of APR techniques proposed [1]–[3], [68]. Accordingly, well-constructed datasets that can be used to evaluate and compare APR techniques are needed and expected to be purified (i.e., without tangled code changes) [31]. However, manual purification is important but time-consuming and error-prone [32]. Furthermore, only Defects4J is manually purified in all datasets that can be used in Java APR [31], and there is no automated approach to alleviate the heavy burden of manual purification, to the best of our knowledge. Therefore, we propose DEPTEST to automate this process, which we expect to be further used in the practical pipelines of dataset construction to provide more accurate ground truth for APR techniques.

VIII. Conclusion

Motivated by the great importance of accurate human-written patches to APR research, we propose an automated dataset purification technique DEPTEST to help automate the dataset purification in the context of test-suite based APR. The DEPTEST is then on Defects4J and achieves statistically significant improvement in terms of the number of purified code lines. Based on the purified data, we further re-dissect the Defects4J to obtain a more accurate understanding of this dataset. We observe the "pseudo multi-chunk bug fixes" during the process, and then systematically revisit multi-chunk bug repair and provide insights for future research.

Acknowledgment

The research was supported in part by the National Key R&D Program of China (No. 2017YFB1001802), the Fundamental Research Funds for the Central Universities (Nos. 2019CDXYRJ0011, 2020CDCGRJ037, 2020CDJQY-A021), and the National Natural Science Foundation of China (Nos. 62002034, 61672529, 61872445).

REFERENCES

[1] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 61–72.

[2] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 173–188.

[3] R. Könighofer and R. Bloem, "Automated error localization and correction for imperative programs," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011, pp. 91–100.

[4] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[5] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[6] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[7] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[8] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 441–444.

[9] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[10] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[11] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 298–309.

[12] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[13] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, 2020, pp. 101–114.

[14] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *2020 ACM/IEEE 42th International Conference on Software Engineering*. IEEE, 2020, pp. 602–614.

[15] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.

[16] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[17] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[18] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "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*, 2019, pp. 302–313.

[19] R. Just, D. Jalali, and M. D. Ernst, "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*, 2014, pp. 437–440.

[20] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "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, 2019, pp. 468–478.

[21] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 10–13.

[22] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[23] D. X. B. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.

[24] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches?: An empirical study on the correct patches generated by automated program repair techniques," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[25] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation." in *ISSTA*, vol. 17, 2017, pp. 226–236.

[26] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" 2020.

[27] P. Cashin, C. Martinez, W. Weimer, and S. Forrest, "Understanding automatically-generated patches through symbolic invariant differences," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 411–414.

[28] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *arXiv preprint arXiv:1909.13694*, 2019.

[29] Z. Y. Ding, Y. Lyu, C. Timperley, and C. Le Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 2019, pp. 2–9.

[30] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2020, pp. 1–8.

[31] S. Herbold, A. Trautsch, and B. Ledel, "Large-scale manual validation of bugfixing changes," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 611–614.

[32] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[33] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 381–392.

[34] G. Gay and R. Just, "Defects4j as a challenge case for the search-based software engineering community," in *International Symposium on Search Based Software Engineering*. Springer, 2020, pp. 255–261.

[35] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.

[36] "Artifact page of our experiment," [Online], available: https://github.com/DehengYang/dataset_purification, 2020.

[37] "Source code of math 65 bug," https://github.com/program-repair/defects4j-dissection/blob/4401e9f/projects/Math/65/org/apache/commons/math/optimization/general/AbstractLeastSquaresOptimizer.java, last accessed: Oct. 2020.

[38] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[39] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[40] "Manual assessment of capgen patches," https://github.com/thwak/confix2019result/blob/master/patches/assessment-others.md#capgen, last accessed: Oct. 2020.

[41] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

[42] A. Zeller, "Yesterday, my program worked. today, it does not. why?" *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6, pp. 253–267, 1999.

[43] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[44] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[45] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–53, 2020.

[46] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017, pp. 55–56.

[47] T. Durieux and M. Monperrus, "Introclassjava: A benchmark of 297 small and buggy java programs," 2016.

[48] M. R. Hoffmann *et al.*, "Jacoco java code coverage library," 2014.

[49] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 378–381.

[50] "Java implementation of delta debugging," https://www.st.cs.uni-saarland.de/whyprogramsfail/code/dd/DD.java, last accessed: Oct. 2020.

[51] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[52] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.

[53] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, 2018.

[54] S. Wang, X. Mao, N. Niu, X. Yi, and A. Guo, "Multi-location program repair strategies learned from past successful experience," *arXiv preprint arXiv:1810.12556*, 2018.

[55] "Manual assessment of math 93 patch of acs," https://github.com/Adobee/ACS#13math93, last accessed: Oct. 2020.

[56] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[57] Y. Yuan and W. Banzhaf, "Making better use of repair templates in automated program repair: A multi-objective approach," in *Evolution in Action: Past, Present and Future*. Springer, 2020, pp. 385–407.

[58] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[59] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[60] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.

[61] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 637–647.

[62] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "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, 2019, pp. 102–113.

[63] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "Lsrepair: Live search of fix ingredients for automated program repair," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 658–662.

[64] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 255–266.

[65] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "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, 2019, pp. 1–12.

[66] C. Artho, "Iterative delta debugging," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 3, pp. 223–246, 2011.

[67] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 134–144.

[68] A. Roychoudhury and Y. Xiong, "Automated program repair: a step towards software automation," *Science China Information Sciences*, vol. 62, no. 10, p. 200103, 2019.