Empirical Software Engineering (2021) 26: 122
https://doi.org/10.1007/s10664-021-10003-7

Where were the repair ingredients for Defects4j bugs?

Exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems

Deheng Yang[1] · Kui Liu[2,3,4] · Dongsun Kim[5] · Anil Koyuncu[6] · Kisub Kim[7] · Haoye Tian[7] · Yan Lei[8] · Xiaoguang Mao[1] · Jacques Klein[7] · Tegawendé F. Bissyandé[7]

Accepted: 8 June 2021 / Published online: 10 September 2021

Abstract

A significant body of automated program repair research has built approaches under the redundancy assumption. Patches are then heuristically generated by leveraging repair ingredients (change actions and donor code) that are found in code bases (either the buggy program itself or big code). For example, common change actions (i.e., fix patterns) are frequently mined offline and serve as an important ingredient for many patch generation engines. Although the repetitiveness of code changes has been studied in general, the literature provides little insight into the relationship between the performance of the repair system and the source code base where the change actions were mined. Similarly, donor code is another important repair ingredient to concretize patches guided by abstract patterns. Yet, little attention has been paid to where such ingredients can actually be found. Through a large scale empirical study on the execution results of 24 repair systems evaluated on real-world bugs from Defects4J, we provide a comprehensive view on the distribution of repair ingredients that are relevant for these bugs. In particular, we show that (1) a half of bugs cannot be fixed simply because the relevant repair ingredient is not available in the search space of donor code; (2) bugs that are correctly fixed by literature tools are mostly addressed with shallow change actions; (3) programs with little history of changes can benefit from mining change actions in other programs; (4) parts of donor code to repair a given bug can be found separately at different search locations; (5) bug-triggering test cases are a rich source for donor code search.

Keywords Automated Program Repair · Fix Ingredient · Code Change Action · Donor Code

1 Introduction

Recently, a momentum of automatic program repair (APR) has been growing in the research community (Britton et al. 2013). The idea of APR is to design algorithms and build code

Communicated by: Saurabh Sinha

✉ Kui Liu
kui.liu@nuaa.edu.cn

Extended author information available on the last page of the article.

transformation pipelines for systematically generating patches that address identified bugs in software code. Three main categories of approaches have been proposed in the literature (Gazzola et al. 2017; Monperrus 2018; Goues et al. 2019; Liu et al. 2021): *heuristics-based* APR techniques leverage heuristics such as genetic mutation operators (Weimer et al. 2009) or fix patterns (Liu et al. 2019a); *constraint-based* APR techniques build on symbolic analysis and constraint solving (Xiong et al. 2017); *learning-aided* APR techniques leverage big code data, e.g., to learn transformation patterns (Gupta et al. 2017).

In general, many of the proposed approaches build on the *redundancy assumption* (Martinez et al. 2014). Redundant code change actions (e.g., insertion of an If statement) can be found in the revision history and can be identified as fix patterns (Kim et al. 2013). The redundancy of code fragments (e.g., identifier tokens such as method names that already exist in the program.) is essential and it has been leveraged by several APR systems as donor code, such as the GenProg (Goues et al. 2012; Weimer et al. 2009; Goues et al. 2012) seminal approach. They together form the *repair ingredients*.

Extensive empirical evaluations on the redundancy assumptions and approaches have been presented in the literature (Liu et al. 2018; Pan et al. 2009; Lou et al. 2019; Sobreira et al. 2018; Martinez and Monperrus 2015; Martinez et al. 2014; Barr et al. 2014), but less work has focused on exploring or validating the adequate locations for optimizing the search of repair ingredients. A study of repetitiveness (Nguyen et al. 2013) shows that 70–100% of small code changes can be found in the revision history. The plastic surgery hypothesis study (Barr et al. 2014) confirms that 43% of changes are graftable from previous changes. At the line-level, 3–17% of commits include code fragments from previous commits (Martinez et al. 2014). However, these studies commonly explored limited scope and granularity. Moreover, the studies did not discuss the impact of repair ingredient locality on the performance of repairing bugs with actual APR techniques.

The redundancy assumption has been leveraged extensively by most program repair approaches (Weimer et al. 2009; Goues et al. 2012; Jiang et al. 2018; Wen et al. 2018; Saha et al. 2017; Liu et al. 2018; Liu et al. 2019a; 2019b; Qi et al. 2014; Qi and Reiss 2017), their design and implementation remain uninformed about a fundamental question: *where are the repair ingredients?* For example, should the APR system leverage fix patterns mined from the buggy program itself? or where is the donor code located for fixing a bug? In addition, an APR tool cannot fix some bugs even if repair ingredients are available. This raises another question: how can we search for repair ingredients? Such questions are particularly acute for generate-and-validate APR systems where defining the location scope of the repair ingredients can have a substantial impact on the quality of the patch candidates. The search space in APR patch generation is indeed determined mainly by two types of ingredients: the change action and the donor code.

Change actions, which are also referred to in the literature as fix templates (Kim et al. 2013), fix patterns (Liu et al. 2019b), abstract modifications (Jiang et al. 2018) or code transforms (Long et al. 2017), are sequences of high-level instructions formulated in a code edit script. Such actions generally target Abstract Syntax Tree (AST) nodes, following the four edite operators that are INSertion, DELetion, UPDate and MOVe. Common examples of change actions in Java APR examples (Liu et al. 2019a) are "*INS CAST checker*", "*UPD If conditional_expression* `exp`" or "*UPD STATEMENT invocation_method* `meth`". Change actions are repair ingredients that are often explicitly constructed in heuristics-based APR. A review of literature approaches reveals three different methodologies to define change actions: (1) *Manual summarization* leverages human-written patches (Pan et al. 2009; Kim et al. 2013) or builds on practitioners' knowledge (Saha et al. 2017; Durieux et al. 2017;

Hua et al. 2018; Qi and Reiss 2017) to define common change actions in bug fix patches. (2) *Statistics of actions* can also be used to systematically (and blindly) consider the most frequent code change actions from a dataset of patches (Jiang et al. 2018; Wen et al. 2018). (3) *Automatic inference* is also implemented to mine from *big-code* some change actions that are more consistent with code contexts or that support different granularities (Long et al. 2017; Liu and Zhong 2018; Liu et al. 2018; Rolim et al. 2018; Koyuncu et al. 2019). To the best of our knowledge, all state of the art approaches in the literature infer change actions from a large dataset that even often excludes the buggy program. Our work investigates the redundancy assumptions of change actions within the buggy program to boost automated program repair towards the momentum of bug self-healing by exploiting the fix ingredients from the buggy program itself (Monperrus 2018).

Donor code is the code fragment (e.g., operator, identifier or expression) that is used in conjunction with the code change action to form the replacement code at the location of the buggy code. For example, when the change action for a buggy code is "*UPD STATEMENT invocation_method* `meth`", the donor code should be an identifier "`new_meth`" that will be used to generate the concrete patch replacing "`meth`" with "`new_meth`". While classical pattern-based repair tools such as PAR (Kim et al. 2013) and TBar (Liu et al. 2019a) explore the buggy file to find the donor code, some recent state-of-the-art systems, such as SimFix (Jiang et al. 2018) and CapGen (Wen et al. 2018) further explore non-buggy code files. Unfortunately, so far, the literature does not provide any comprehensive assessment of the impact of the configuration of donor code search scope on the performance gain.

In this paper, we perform an empirical study on exploring the impact of repair ingredient retrieval on the performance of 24 APR tools. We first conduct a post-mortem analysis on where the repair ingredients can be found. Based on the analysis, we explore how the locality of repair ingredients impacts the performance of APR. Although diverse benchmarks of bugs have been proposed for APR (Durieux et al. 2019), the data of other benchmarks are not comparable to that of Defects4J in terms of popularity. Therefore, we select Defects4J, which is regarded as the most commonly used benchmark and has the largest number of repair results by existing APR tools, for obtaining reliable results.

The main insights of this study are:

1. **Availability of Change Actions:** For 70% of bugs in Defects4J, the necessary change actions from the local revision history cannot be discovered. This implies that the size of a local revision history might not be sufficiently large to avoid searching in external sources for change actions.
2. **Availability of Donor Code:** Necessary donor code to fix a bug is available in the buggy program for a half of the Defects4J benchmark bugs, which is located in the different scales of search space (even in the bug-triggering test cases). However, it is necessary to explore external sources to collect donor code to fix the other half of Defects4J bugs.
3. **Impact of Change Actions on the Performance of APR Systems:** APR systems tend to be more effective if a given bug can be fixed with simple change actions (where the number of change actions < 3 and the depth of change actions < 6). This pushes the APR community to put more efforts for complicated bugs.
4. **Impact of Donor Code Locality on the Performance of APR Systems:** APR systems are generally more effective when donor code is available in the buggy program and when it is near to the bug location. There should be a higher probability to fix when exploring a larger search space but it takes more time and cost before finding successful patches.

2 Background & Research Questions

Automated program repair consists of three basic steps: fault localization, patch generation and patch validation. While recent work increasingly investigates fault localization (Qi et al. 2013; Xuan and Monperrus 2014; Yang et al. 2017; Just et al. 2018; Yang et al. 2021) and patch validation (Koyuncu et al. 2019; Qi et al. 2015; Long and Rinard 2016; Xiong et al. 2018), patch generation approaches (Wen et al. 2018; Liu et al. 2019; Xiong et al. 2017; Jiang et al. 2018; Le et al. 2016; Liu et al. 2019b; Saha et al. 2017) have been the main focus of APR research. The overall objective of each approach is to optimize the selection of relevant repair ingredients in order to generate correct patches (i.e., adequacy of repair ingredients is key) for the largest number of bugs (i.e., the diversity of repair ingredients is key). Many approaches rely on specified or mined change actions to guide the patch generation. For example, GenProg (Weimer et al. 2009; Goues et al. 2012) considers two genetic operators (*mutation* and *crossover*) as the change actions. PAR (Kim et al. 2013) was subsequently proposed to include common change actions in human-written patches. Since then, recurrent code change actions (i.e., fix patterns (Liu et al. 2018; Liu et al. 2019b), fix templates (Liu et al. 2019a; Liu et al. 2018; Saha et al. 2017), code transformations (Long et al. 2017), etc.) have been widely explored in the APR literature (Wen et al. 2018; Jiang et al. 2018).

Where are then the repair ingredients? One of the common approaches in the literature that was initiated by Kim et al. (2013) and later systematically applied, is to mine recurrent code change actions from large datasets of program revisions. Nguyen et al. (2013) investigated the repetitiveness of atomic change actions, but ignored the availability of complete change actions for a bug fix, which is critical to examine the impact on the performance of APR. The complete change action is associated with complete AST context path and the corresponding change operations for fixing bugs. For instance, as shown in Fig. 1, the change action of the Closure-10 bug fix that consists of 3 change operations (i.e., op_1, op_2, and op_3, `UPD-ReturnStatement` → `UPD-MethodInvocation` → `UPD-MethodName`) is a complete change action. On the contrary, the atomic change action denotes the single change operation worked on the leaf AST node without any context (i.e., op_3, `UPD-MethodName` in Fig. 1), which is not accurate enough for change action selection in program repair (Liu et al. 2019a). Thus, we ask the following research question about change actions:

– **RQ-1:** ***To what extent change actions available and accessible in the buggy program are relevant to repair it?*** We revisit the availability of change actions from the buggy program history for APR. Specifically, we assess not only the availability of change actions, but also whether they are found (i.e., are they redundant enough to be considered as candidate change actions?) by APR approaches and the underlying reasons of the high or low availability of change actions.

Concretely, once a change action is selected, the patch generation requires some donor code to synthesize the actual replacement code. Common examples of donor code are variables (e.g., adding a missing NULL pointer check requires to identify which variable to check) or identifiers (e.g., replacing a wrong API method call requires another method name). Finding relevant donor code is thus a challenging endeavour in program repair. Unfortunately, this challenge can be hidden as an engineering detail, and is thus scarcely discussed in the literature. APR research however must strike a good balance between efficiency (i.e., the search space must not explode by considering all possibilities) and

```
--- src/com/google/javascript/jscomp/NodeUtil.java
+++ src/com/google/javascript/jscomp/NodeUtil.java
@@ -1416,3 +1416,3 @@ static boolean mayBeString(Node n) {
    if (recurse) {
-       return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
+       return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
    } else {
```

Change Action:
op_1: (UPD, ReturnStatement, null, {op_2})
op_2: (UPD, MethodInvocation, op_1, {op_3})
op_3: (UPD, MethodName, op_2, ∅)

Donor Code: ‘‘anyResultsMatch’’ for op_3

Fig. 1 Example of change action and donor code from the patch of fixing bug Closure-10 in Defecst4J

effectiveness (i.e., the search space must include the relevant donor code to yield correct patches). For example, Kim et al. (2013), in their implementation of PAR, have focused on the local buggy file to search for donor code candidates. TBar (Liu et al. 2019a) has followed on this assumption that the buggy file is enough. Xiong et al. (2017), however, leverage constraints to extract donor code from the bug-triggering test cases. SimFix (Jiang et al. 2018) and CapGen (Wen et al. 2018) leverage all files (including the non-buggy ones). A more extensive search is performed in LSRepair (Liu et al. 2018) where the authors propose to perform a live search of large code bases of real-world programs (e.g., GitHub repositories).

Different spaces for donor code search have been explored in the literature, but the community has not separately investigated the added value of where the donor code is likely located. In Martinez et al.'s work (Martinez et al. 2014), the granularity of redundancy is investigated only in two types of scope: local file or whole project. Motivated by GenProg (Weimer et al. 2009), the *plastic surgery hypothesis* (Barr et al. 2014) explores the redundancy of the concrete donor code snippets, and briefly mentions file and package-level locality. However, the two studies have limitations on the granularity and search space of donor code, which are not sufficient to figure out the impact of donor code availability on the repair performance (i.e., number of plausibly or correctly fixed bugs) of APR. To address this limitation, we consider a more comprehensive granularity and search space of donor code (cf. Section 3.2) for fixing bugs. The following research question quantifies the details of the spatial locality of donor code in program repair.

– **RQ-2:** ***Where could the donor code be found to generate patches?*** Heuristics-based, constraint-based and learning-aided repair approaches explore various advanced methods to optimize the search and selection of donor code within the buggy file (Liu et al. 2019b; 2019a), non-buggy files (Wen et al. 2018; Jiang et al. 2018), the test cases of the buggy program (Xiong et al. 2017), and other programs (Liu et al. 2018). For the first time in the literature, we propose a comprehensive study on Defects4J to clarify which search scopes are adapted towards an effective program repair system.

Finally, given the lack of focused empirical validation of search strategies for repair ingredients, we propose to investigate the current APR systems and their yielded patches that make the program pass all test suites.

– **RQ-3:** ***To what extent the locality of repair ingredients impacts the repair performance of APR systems?*** A number of APR studies have already expressed the importance of repair ingredients on repair performance (Qi and Reiss 2017; Le et al.

2016). However, assessment results in the literature often elude the contribution of the novelty in repair ingredient search. We propose to comprehensively investigate the impact of the locality of repair ingredients on the repair performance of 24 state-of-the-art APR systems.

3 Definitions

This section clarifies the notions of repair ingredients related to change actions and donor code presented in this work.

3.1 Change Actions

In this study, we define a change action (ca_i) as a hierarchical sequence of change operations ($\{op_1, op_2, op_3, \ldots, op_n\}$) presented in a tree structure. The change operations are expected to be applied to the abstract syntax tree (AST) of the given buggy code. Each operation can be represented by a tuple of (1) change instruction, (2) code element type where the operation is applied to, (3) parent operation, and (4) a set of child operations. These can be defined as:

$$\begin{aligned} ca_i &= \{op_1, op_2, op_3, \ldots, op_n\}, \text{ with} \\ op_k &= (inst, e, op_p, \{OP_c\}) \end{aligned} \tag{1}$$

where a change action ca_i is a set of change operations structured as a tree. In each change operation op_k, the first element denotes a change instruction, which is one of **update**, **insert**, **delete**, and **move** (i.e., $inst \in \{UPD, INS, DEL, MOV\}$ which are classical operations for code differencing tools, e.g., GumTree (Falleri et al. 2014)).

The second element (i.e., e) in op_k is the context, determined by the AST node type where op_k could be applied for. For example, e can be `ReturnStatement`, `Expression`, and `VariableDeclaration`. op_p represents the parent operation of the current operation. op_p is `Null` if the operation is the root node of ca_i. OP_c is a set of child operations of the current operation. OP_c is an empty set if the operation is a leaf node in ca_i.

Consider the example patch illustrated in Fig. 1 for bug Closure-10 in Defects4J. This patch implements a unique change action, which consists of three change operations (i.e., three *UPD* instructions) applied to three hierarchical code elements: `ReturnStatement`, `MethodInvocation`, and `MethodName`. It starts with updating a `ReturnStatement` of which $\{OP_c\}$ is to update a `MethodInvocation` element. The final operation is to replace the buggy `MethodName` of the `MethodInvocation` with another `MethodName`. Such an abstraction of code change action allows its application to any other buggy statements having the same AST contexts (i.e., `ReturnStatement` → `MethodInvocation` → `MethodName`). In other words, the abstraction of the change action not only records the change patterns of the bug fix, but also preserves the AST context surrounding the bug. It discards the specific tokens (e.g., identifiers and operators) in the concrete patch to ensure its applicability for other bugs with the same AST contexts.

Note that, in this study, multiple changes that share the same root change operation are regarded as a single change action. Conversely, changes with different root operations are distinguished as independent change actions. Among all AST node types, only

{*}Statement[1] can be a root operation in a change action. Due to the nested structure of programs, a given change action may affect several statements. In such a case, the root of a change action is the parent statement that is nearest to the leaf node. For example, in Fig. 1, the root of the change action is the ReturnStatement rather than the IfStatement (which is the super-parent Statement node) or the MethodInvocation (which is not a Statement node).

To fix a single bug, several change actions can be necessary for some cases. Our study, of course, explores those cases and investigates how they affect the results of APR systems. The results are reported in Section 5.

3.2 Donor Code

Donor code is a code element that is used to synthesize (e.g., replace or insert) a patch candidate. It is represented as:

$$dc_j \in C_{blank} \cup C_{frag} \tag{2}$$

where C_{blank} denotes the case where the donor code is not required in a change operation (e.g., a deletion operation). However, a change action is generally applied by leveraging a specific donor code in C_{frag}, which is a set of code fragments. A code fragment can be any program element including variables in a program. Code fragments are thus operators (e.g., $+$, $-$, and $*$), identifiers, expressions, statements, and even blocks of code. As sources, donor code can be identified locally (i.e., within the program where the bug has been found) or can be mined from external programs (e.g., programs in open-source projects). For example, the change action shown in Fig. 1 requires a donor code (i.e., the method name "anyResultsMatch") for op_3 to replace the bug-related element (i.e., method name "allResultsMatch").

In this work, we further categorize the donor code into 7 categories following the location where it is sourced:

1. **Intrinsic Donor Code**: such donor code is defined as native tokens in a program language specification. These include all operators and program keywords (e.g., basic data types and control structures).
2. **Local-Method Donor Code**: the donor code can be retrieved from the method code where the bug was localized.
3. **Local-File Donor Code**: the donor code can be retrieved from the file where the bug was localized. However, in this study, we exclusively consider only donor code that is within the file but not within the buggy method.
4. **Local-Package Donor Code**: the donor code can be retrieved from the buggy package where the buggy file is located. Again, we consider only cases where the donor code is not in the buggy file.
5. **Local-Program Donor Code**: the donor code can be found within the buggy program source code, but not in the buggy package or test code.
6. **Local-Test Donor Code**: the donor code can only be found within the code of the bug-triggering test cases.
7. **Global Donor Code**: the donor code cannot be found in the buggy program nor the test cases, but could be found in other programs.

[1] ReturnStatement, IfStatement, etc.

Table 1 Defects4J dataset information

	Chart	Closure	Lang	Math	Mockito	Time	Total
# Bugs	26	133	65	106	38	27	395

Note that a code fragment (including the donor code tokens) might be available at multiple categories (e.g., from both intrinsic and local-method), we group such fragment into the smallest category where it can be identified (i.e., the intrinsic donor fragment in the above example). That is, an APR tool does not need to expand the donor code search into a larger scope (e.g., local-method in this example) for fixing a bug. In addition, for a bug that requires multiple donor code fragments (including the donor code tokens), its fragments could be found at different categories (e.g., one at local-method and another at local-file level). In such cases, we classify the donor code of the bug into the largest category where the donor fragments can be found (i.e., the local-file in this example) since an APR system has to search all donor code in the local-file level.

4 Study Design

4.1 Subjects

To answer our research questions, we conduct a study on the Defects4J dataset (Just et al. 2014). This dataset was selected since it has been leveraged by most APR tools (Le et al. 2016; Xiong et al. 2017; Saha et al. 2017; Qi and Reiss 2017; Martinez and Martin Monperrus. 2016; Xuan et al. 2017; Martinez et al. 2017; Chen et al. 2017; Liu and Zhong 2018; Jiang et al. 2018; Wen et al. 2018; Hua et al. 2018; Liu et al. 2018; Liu et al. 2019b; 2019a; Saha et al. 2019). Defects4J is a curated dataset, where the bugs and their fixes are clearly isolated from other changes. Table 1 summarizes the number of bugs in the version 1.4.0 (GitHub 2021) of Defects4J.

As subject APR systems, we consider 24 tools from the literature, which are evaluated on the Defects4J and have been reviewed by peer researchers. When investigating APR performance, we leverage the reported results by these tools. The column "*Repair Results*" in Table 2 indicates the number of bugs fixed by each APR system; the numbers outside of parentheses are the number of correctly fixed bugs while the numbers in the parentheses indicate the number of plausibly-fixed bugs. We recall that a plausible patch is a patch that makes the program pass all test cases. A correct patch is a patch that is equivalent to the patch provided by developers in the benchmark. Correctness is decided manually by the authors (Qi et al. 2015).

4.2 Identification of Change Actions

To investigate the availability of change actions in the buggy program history (specifically for RQ-1), it is necessary to compare the actual patch of a given bug and all available bug-fixing changes in the revision history[2] of the associated program. Defects4J provides the necessary ground truth information on the actual patches. We then collect other

[2]The revision history may not be able to explore for some projects but most APR studies (Martinez et al. 2014; Pan et al. 2009; Barr et al. 2014; Nguyen et al. 2013; Liu et al. 2018; Zhong and Su 2015) assume that

Table 2 APR systems evaluated with Defects4J

APR System	Authors	Publication Venue	Publication Year	Repair Results
HDRepair (Le et al. 2016)	Le et al.	SANER	2016	6 (23)
jGenProg (Martinez and Martin Monperrus. 2016)	Martinez and Monperrus	ISSTA	2016	5 (27)
jKali (Martinez and Martin Monperrus. 2016)	Martinez and Monperrus	ISSTA	2016	1 (22)
jMutRepair (Martinez and Martin Monperrus. 2016)	Martinez and Monperrus	ISSTA	2016	3 (17)
Nopol (Xuan et al. 2017)	Xuan et al.	IEEE TSE	2017	5 (35)
ACS (Xiong et al. 2017)	Xiong et al.	ICSE	2017	18 (23)
ELIXIR (Saha et al. 2017)	Saha et al.	ASE	2017	26 (41)
JAID (Chen et al. 2017)	Chen et al.	ASE	2017	9 (31)
ssFix (Qi and Reiss 2017)	Xin and Reiss	ASE	2017	20 (60)
LSRepair (Liu et al. 2018)	Liu et al.	APSEC	2018	19 (37)
CapGen (Wen et al. 2018)	Wen et al.	ICSE	2018	21 (25)
SketchFix (Hua et al. 2018)	Hua et al.	ICSE	2018	19 (26)
SimFix (Jiang et al. 2018)	Jiang et al.	ISSTA	2018	34 (56)
ARJA (Yuan and Banzhaf 2018)	Yuan and Banzhaf	IEEE TSE	2018	18 (59)
SOFix (Liu and Zhong 2018)	Liu and Zhong	SANER	2018	23 (23)
FixMiner (Koyuncu et al. 2019)	Koyuncu et al.	EMSE	2019	25 (31)
kPAR (Liu et al. 2019)	Liu et al.	ICST	2019	18 (49)
AVATAR (Liu et al. 2019b)	Liu et al.	SANER	2019	27 (53)
TBar (Liu et al. 2019a)	Liu et al.	ISSTA	2019	43 (81)
PraPR (Ghanbari et al. 2019)	Ghanbari et al.	ISSTA	2019	43 (148)
Hercules (Saha et al. 2019)	Saha et al.	ICSE	2019	46 (63)
VFix (Xu et al. 2019)	Xu et al.	ICSE	2019	12 (15)
GenPat (Jiang et al. 2019)	Jiang et al.	ASE	2019	16 (42)
ConFix (Kim and Kim 2019)	Kim and Kim	EMSE	2019	22 (92)

bug-fixing changes by using keyword matching (namely "bug", "error", "fault", "fix", "patch" or "repair"); this approach is largely used in the literature (Liu et al. 2018; Le et al. 2016; Zhong and Su 2015; Pan et al. 2009; Mockus and Votta 2000). For each Defects4J bug, we consider only the subset of history changes that were committed before the bug's fix commit. Eventually, all changes (Defects4J patches and other bug-fixing changes) are processed with the following two steps to identify actual change actions in each change set:

1. For each change, checkout from the repository the buggy version and the fixed version of the program.
2. Extract the relevant code change actions in the AST tree differencing representation. We leverage in this study the `PatchParser`[3] open source tool proposed by Liu

the revision history is available. The revision history in our study is also available as Defects4J bugs are all collected from real-world and large-scale projects.

[3] https://github.com/AutoProRepair/PatchParser.git

et al. (2018). `PatchParser` builds on the GumTree (Falleri et al. 2014) and provides a hierarchical structure (which includes the whole code context of the change).

4.3 Donor Code Identification and Search

Identifying the ground truth of donor code To assess the locality of donor code, we first identify the ground truth of what each bug fix uses as donor code in Defects4J. For practicality of analysis, we leverage the representation provided in the artefact release[4] of a previous study by Sobreira et al. (2018). In our work, the donor code search is conducted in a fine-grained way, e.g, statements, expressions, method names (the example in Fig. 1), variable names, data types and so on, which can be directly used to synthesize patch candidates with change actions in APR systems.

To determine donor code availability, we leverage the code clone detection technique (Roy et al. 2009) to search a potential code fragment that is matching the donor code associated to Defects4J patch. Specifically, we employ the tree-based clone detection technique (i.e., Deckard) proposed by Jiang et al. (2007) to detect type-1 and 2 clones, which has also been used by existing APR systems (Le et al. 2016; Jiang et al. 2018). Thus, two types of code clone are employed in this paper:

- **Type-1**: Identical code fragments except for variations in whitespace, layout and comments.
- **Type-2**: Syntactically identical code fragments except for variations in identifiers, types, whitespace, layout and comments.

Algorithm 1 Identifying the location of donor code candidates.

Input: target buggy program: $Prog$.
Input: exact donor code (ground truth): Edc.
Output: nearest position of candidates:

```
Pos ∈ {Intrinsic, Method, File, Pacakge, Program, Global}.
function IDENTIFYNEAREST(Prog, Edc)
    candidates := traverseFragmentsInProg(Prog).map(frag → matchAndLocalize(Edc, frag));
    Pos := nearest(candidates);
end function
function MATCHANDLOCALIZE(Edc, frag)
    Pos := Global;
    if Edc is an intrinsic donor code then
        Pos := Intrinsic;
    else if isType1Match(Edc, frag) || isType2Match(Edc, frag) then
        if getMethod(Edc) is getMethod(frag) then
            Pos := Method;
        else if getFile(Edc) is getFile(frag) then
            Pos := File;
        else if getPackage(Edc) is getPackage(frag) then
            Pos := Package;
        else
            Pos := Program;
        end if
    end if
end function
```

[4] https://github.com/program-repair/defects4j-dissection/blob/master/defects4j-patch.md

Search for Donor Code Given the required donor code for a Defects4J bug-fixing change, we search for a set of donor code candidates and identify their location categories (cf. Section 3.2). Algorithm 1 describes the search process. For all code fragments of the production code in the program (Line 3)[5], we first examine whether the donor code is "intrinsic" (Line 8) or blank. Otherwise, donor code candidates can be found somewhere in the program where the bug has been found (Line 10). If candidate donor code fragments are found in the program (Line 10), the algorithm checks whether they are located in the buggy method (Line 12), buggy file (Line 14), buggy package (Line 16), or another location in the production code (Line 18). Since there could be several donor code candidates, the algorithm takes one that is nearest to the bug location (Line 4) (i.e., in the sequence of `Intrinsic`, `Method`, `File`, `Package`, `Program`, and `Global`). For example, the donor code (`anyResultsMatch`) shown in Fig. 1 can be found in several locations in the program, with the nearest being located in the same '`File`'.

Besides the production code, our study also scans the bug-triggering test cases for donor code. This may provide an insight of whether test cases are a valuable source of donor code, since several state of the art approaches mention them in their implementation details but do not comprehensively assess the added-value of such an extraction source. This information is investigated independently against the location information of the production code above.

5 Study Results

We provide experimental data towards answering the relevant research questions of this study (cf. Section 2). Key insights are further highlighted to inform future development of APR. In Section 5.1, we focus on answering **RQ1** by investigating to what extent change actions can be found in the buggy program's commit history (i.e., validating the redundancy assumption for the Defects4J bug fixes). Section 5.1 also draws a landscape of properties of change actions horizontally (in terms of quantity per patch) and vertically (in terms of depth distribution), which provides data for further analysis of Section 5.3. In Section 5.2, we report on the localization of potential donor code fragments in response to **RQ2**. Finally we investigate the relationship between APR performance and the location of repair ingredients in Section 5.3 and then discuss the performance impact of the location where donor code can be found (i.e., **RQ3**).

5.1 RQ-1: Availability of Change Actions

We investigate the availability and complexity of change actions.

Availability of Change Actions The relevant literature of APR generally builds change action databases by mining code bases that are independent of the buggy program. We propose to investigate this redundancy assumption of change actions for the Defects4J bugs to verify whether change actions can be mined from the buggy program itself. Note that, because we could not associate the benchmark-provided bug fix of Chart-6, Chart-11 and Chart-26 to a precise commit in the JFreeChart repository, we exclude them.

Figure 2 summarizes the proportion of bugs for which the fixes are implemented with change actions that are available in the buggy program history. The tag "*available*" indicates

[5]In contrast with the mining of code change actions, most APR systems scan only the production code of a program for donor code.

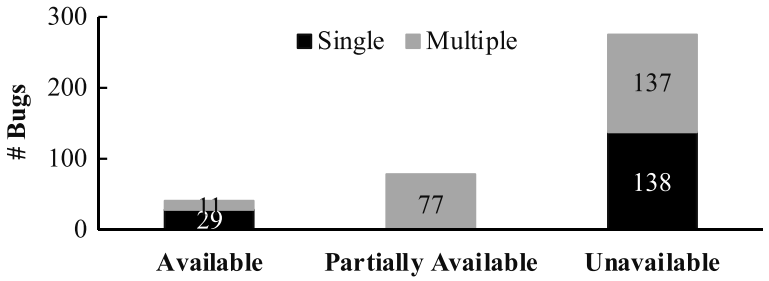


Fig. 2 Available results of bugs comprising single and multiple change actions

that all needed change actions can indeed be found in the buggy program history. "*Partially Available*" indicates that only some of the required change actions for a given bug can be retrieved in the buggy program. "*Unavailable*" indicates that none of the change actions could be extracted from the program history at the time of fix. "*Single*" and "*Multiple*" labels further help to perceive the proportion of cases where the required change actions set for a bug was a singleton (i.e., Single) or not (i.e., Multiple).

We observe that among the 167 (=29+138)[6] cases where a single change action is required, only 21% (29/167) of the change actions were available in the history of the associated program. This puts into perspective the redundancy assumption (Martinez et al. 2014): the buggy programs in the Defects4J dataset do not have in their history all necessary change actions to drive the patch generation.

►**RQ-1.1** ◄ *Why are change actions not available in the program history ?*

Our conjecture is that change actions can be available only if the buggy program has a long history of code changes. Indeed, statistically the more changes (regardless of bug fixing or non-bug fixing commits) are performed, the more chances to encounter a recurring repair action. The distributions of historical commits presented in Fig. 3 are indeed consistent with the distributions of bugs for which change actions are available in the history of the program (cf. Fig. 4). Closure presents the largest median number of commits that form the history of the program before the fix associated to Closure Defects4J bugs. It also has the highest number of cases where the change action is available in the program history.

The redundancy assumption is constrained by the size of available historical change data for mining recurrent change actions.

►**RQ-1.2**◄ *Are the available change actions most recurrent in the buggy program history?*

Change actions are mined in the APR literature (Kim et al. 2013; Le et al. 2016; Wen et al. 2018; Jiang et al. 2018) based on their frequency. Although our aforementioned sub-question confirmed that some relevant change actions can be found in the history of a buggy

[6]This number is different from 169 shown in Fig. 6 since the revisions related to Chart-11 and 26 are not available to explore as stated in the above paragraphs.

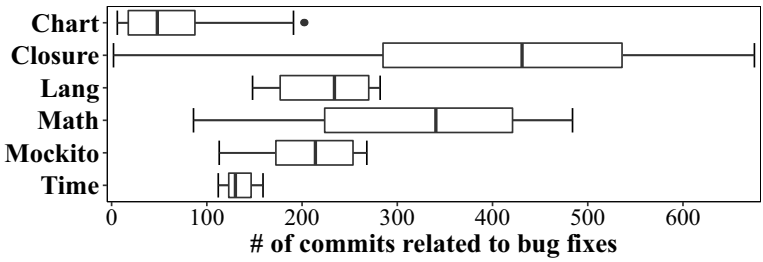


Fig. 3 Distribution of the number of commits related to bug fixes in each project

program, it is not guaranteed that such a change action could have been considered in a statistical pattern mining approach.

We investigate the relative redundancy of Defects4J bug fix actions that are available in the program history. For each relevant change action we compute its ranking in the list of change actions, which are sorted by their frequency of appearance in the history of the buggy program changes.

Figure 5 provides the distribution of bugs for which the relevant change action is within some ranking ranges. Note that when a bug is fixed by multiple change actions, we consider that the least frequent change action limits the likelihood of successful mining. To reflect this, we always consider the lowest ranking, among the rankings of all needed change actions, as the final ranking associated to the change actions associated to a bug.

When we consider fully available change actions, we can count only one bug in Defects4J whose change action is among the top-10 most recurrent in the history of the associated buggy program. The overall observation is that the relevant change action is actually among the least recurrent in the buggy program (average frequency ranking of 335 for available change actions).

Relevant change actions for repairing Defects4J programs are generally far from being the most frequent change actions in the program's historical changes. This finding suggests that Defects4J bugs may not be of the most common types in the associated projects.

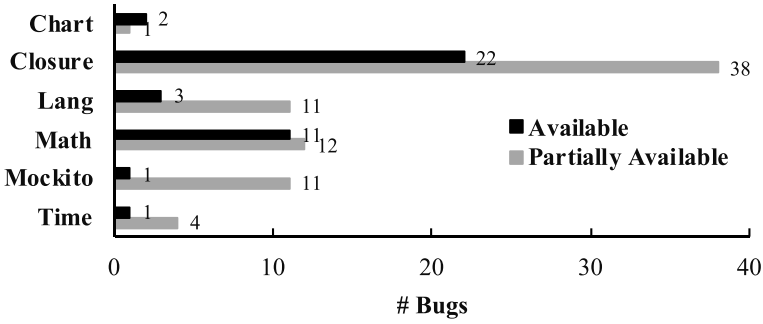


Fig. 4 Number of bugs in which necessary change actions are available in the buggy program history

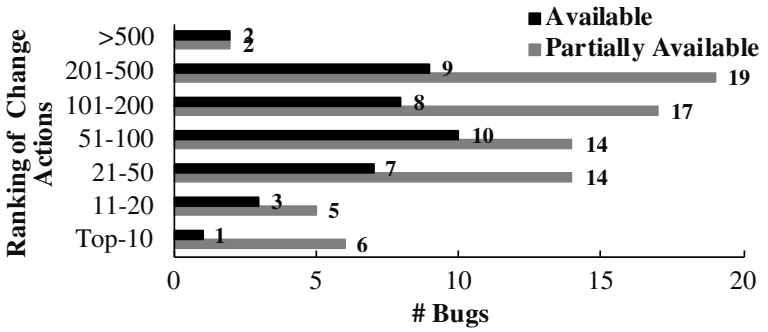


Fig. 5 Rankings of available change actions from the buggy programs themselves

►**RQ-1.3**◄ *Are the availability of change actions related to their complexity?*

Complexity of Change Actions The complexity of bugs is computed in the literature based on the number of files and lines that are changed by patches. Motwani et al. (2018) have used such a computing approach to assess the complexity of Defects4J bugs that were successfully addressed by APR tools. Their conclusion was that "automated repair is more likely to patch easy defects than hard ones, reducing its utility". While this definition is commonly accepted, it may hide a simple fact that a single line change may actually be a complex change, depending on the depth of change actions that must be applied. For example, a fix that requires changing variables and operators simultaneously may be hard to engineer. Thus, we propose to investigate the complexity of Defects4J fixes in the number of change actions (e.g., change the condition in a For loop vs change the condition in a For loop and change the initialization of the iteration variable), as well as a more fine-grained manner based on the vertical depth of the required change actions (e.g., delete an IfStatement vs delete a specific sub-expression of the conditional expression in the IfStatement), to further assess the relationship between the availability of change actions and their complexity.

Figure 6 presents the distribution of the number of change actions within Defects4J bug fixes. A significant share of Defects4J bugs (i.e., ($\sim$75%=$\frac{169+86+42}{395}$)) require three or fewer change actions to fix them. An example of such a bug is Closure-10 whose fix was presented in Fig. 1. We also observe that most available change actions mainly have less number of change actions.

We further define the depth of a change action ca as:

$$\max_{\forall op_j \in ops(ca)} distance(root(ca, op_j)) \tag{3}$$

where $ops(ca)$ enumerates all change operations of a given change action ca. The depth reflects to what extent the change action targets a fine-grained entity of the AST graph at the buggy code location. Figure 7 illustrates the relevant AST of the buggy code of Closure-10 (cf. Fig. 1). The depth of the relevant change action to apply for this bug is 3 (as denoted by the path with red circles).

Figure 8 provides the statistical distributions of the depths of change actions for fixing Defects4J bugs. Note that in this study we consider, for each bug, the maximum of the depths of different change actions involved in the fix of that bug. We note that only a few bugs can be fixed with shallow change actions (e.g., directly deleting the buggy statement).

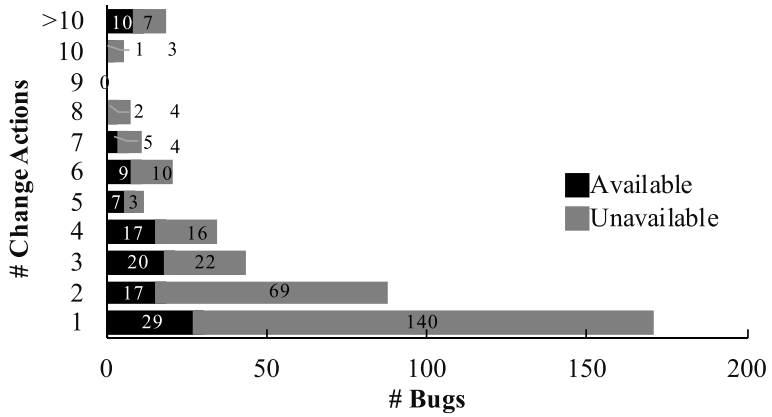


Fig. 6 Distribution of the numbers of change actions for fixing Defects4J bugs

68.4% (= $\frac{65+78+72+55}{395}$) of bugs are fixed by targeted change actions going down at depths 4–7 in the AST of the buggy code location, of which depths occupy most available change actions. This result puts into perspective the reality of the complexity of Defects4J bugs. If complexity is measured by the challenge of digging into the AST of the buggy code to identify a specific code element to modify for fixing the bug, then a large proportion of Defects4J bugs can be said to be complex. This finding validates the assumptions by recent research efforts (Jiang et al. 2018; Koyuncu et al. 2019) in the literature towards building approaches that mine fine-grained fix patterns (i.e., that target deeper nodes in an AST context) to improve repair accuracy.

Defects4J bugs mostly require a limited number of change actions, but often target deep nodes in the AST sub-tree of the buggy code. And most available change actions present simple complexity. It requires APR researchers to carefully mine repair ingredients for generating correct patches.

5.2 RQ-2: Availability of Donor Code

We now focus on exploring the location of donor code to concretize the fix patches.

Locations of Donor Code Fig. 9 illustrates the distributions of bugs associated to locations where a code fragment was found to be a type-1 or type-2 clone of the necessary donor code used to generate the bug fix patch. Specifically, the Y-axis counts the number of bugs that can be fixed by searching for at most local-**X** (e.g., local-method or local-file) level. It

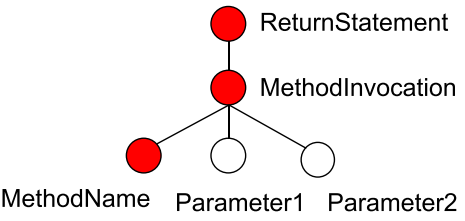


Fig. 7 The AST of Defects4J bug Closure-10 impacted by its change action

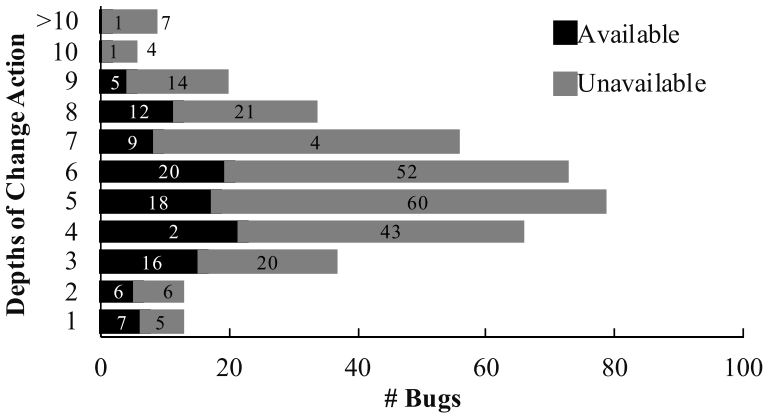


Fig. 8 Distribution of the depths of change actions for fixing Defects4J bugs

appears that the donor code is often intrinsic (i.e., simple programming constructs such as operators). When the donor code is available in the program it is generally located within the buggy method or the buggy file. In only a few cases, it is in other parts of the program.

Note that Fig. 9 data includes only cases of 196 bugs where a set of donor code can be discovered within the production code. For the 199 ($= 395 - 196 \sim 50\%$) remaining bugs, the donor code should be searched outside the buggy program (e.g., Chart-6). Unfortunately, enlarging the search scope for donor code will lead to an explosion of patch candidates, leading to a poor correctness ratio (Wen et al. 2018) (i.e., increasing probability to generate plausible but incorrect patches).

Half of Defects4J bugs require some donor code, which are outside the buggy program. Conversely, for the other half, a donor code fragment is generally located near the buggy code location.

Repair Potential with Intrinsic Donor Code Our investigation data reveals that 44 bugs could be actually repaired by using programming language constructs such as operators as donor code. Such constructs are generally manipulated by simple genetic improvement operations. Yet, only a single APR system, to date, has been able to fix over 44 bugs in the Defects4J dataset. Unfortunately, a closer look at Hercules (Saha et al. 2019) patches reveals that these are not even associated to all the above 44 bugs.

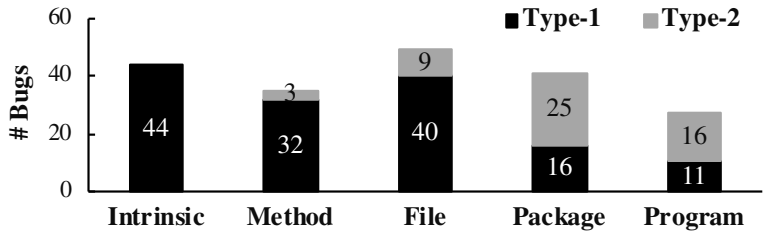


Fig. 9 Distributions of donor code categories

Although searching for relevant fix donor code is important, it is noteworthy that state-of-the-art APR systems still fail to fix some bugs which do not require donor code search in a big search space.

Availability of Donor Code Following up on the discussion above, which focused on individual donor code fragment, we consider all necessary code fragments for fixing a given bug, and investigate whether they can be retrieved in the same search scope. This dissection further considers cases where "All" donor code fragment can be found in the buggy program (In Fig. 10, 155 + 45 = 196 + 4, where the donor code of 4 bugs is available from the bug triggering test cases), or only "Partial" donor code fragments can be found in the buggy program, or the donor code is "Global" (i.e., fully outside the buggy program).

Figure 10's data reveal that, while a large share of bugs have donor code fragments that are all available in the "Same locations", for many bug cases the donor code fragments are dispersed across different locations, which challenge the possibility for effective repair. For example, to generate a valid patch for Chart-2, a part of donor code ("`minimum = Math.min(minimum, value);`") can be found in the buggy method, the other part ("`value = intervalXYData.getXValue(series, item);`") can be identified from the buggy class file but not the buggy method, and the remaining donor code can be retrieved from the other code files that are not in the buggy package.

Donor code fragments can be dispersed across different locations within the buggy program and outside the program. This finding justifies why some state-of-the-art APR systems, which focus the donor code search to a specific location, manage to only partially fix some of the Defects4J bugs. They could only identify some of the donor code fragments.

Availability of Donor Code in Bug-Triggering Test Cases Our study has produced data that suggest that donor code is also available in the bug-triggering test cases for 38 bugs. For 13 bugs, all donor code fragments are in the test cases, while for other 25 bugs only some of the necessary code fragments can be found in the test cases. The ACS (Xiong et al. 2017) repair system indeed leverages information in bug triggering test cases for precise condition synthesis of patches. Our study further validates for the whole research community the added value of leveraging test cases as a source for donor code search.

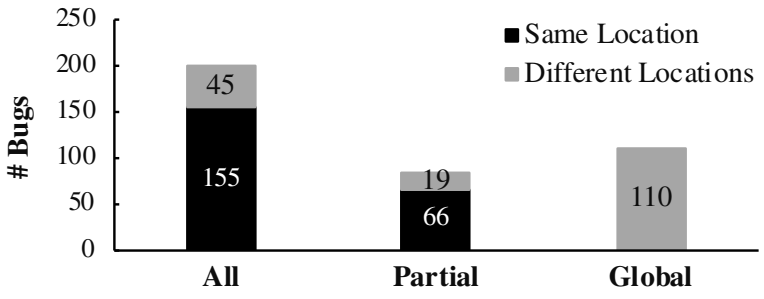


Fig. 10 Overview availability of donor code

Nevertheless, this finding must be put into perspective in practice. In a recent study, Koyuncu et al. (2019) discussed the availability of bug-triggering test cases and highlighted that such test cases are often unavailable at the time that bugs are reported by end-users. All Defects4J bugs are curated with all relevant test cases (Just et al. 2014). If removing the future test cases (i.e., bug-triggering test cases provided when submitting patches (Koyuncu et al. 2019)) from Defects4J bugs, the necessary code fragments only for four bugs can be retrieved from the test cases. The repair system will therefore fail to find the adequate donor code. Therefore, when the bug-triggering test cases are unavailable, it will be not actionable for retrieving the donor code from test cases.

Bug-triggering test cases are an important target for searching donor code, but their availability could arise a new challenge for donor code searching.

5.3 RQ-3: Impact of Repair Ingredients Locality on APR Performance

We explore the impact of repair ingredient properties required to repair Defects4J buggy programs (cf. Sections 5.1 and 5.2) on the performance of APR systems in the literature. In particular, we stress on the quantitative differences between bugs that were eventually correctly-fixed by at least one APR tool, those that are only plausibly (but incorrectly) fixed by some APR tools, and those that remain unfixed to date by any APR tool.

►**RQ-3.1**◄ *To what extent the complexity of change actions required for Defects4J bugs can impact the repair performance of APR systems?*

To answer RQ-3.1, we consider a careful review of reported results in the literature. Hence we focused on Defects4J version that was used by 24 APR systems (listed in Table 2) for evaluation. Figure 11 provide statistics about the impact of code change actions breadth on APR performance.

Overall, we note that state-of-the-art APR systems successfully produce correct/plausible patches for bugs that require few change actions (here, # of change actions < 3). Additionally, the state-of-the-art APR systems can only fix a small number of complex bugs that

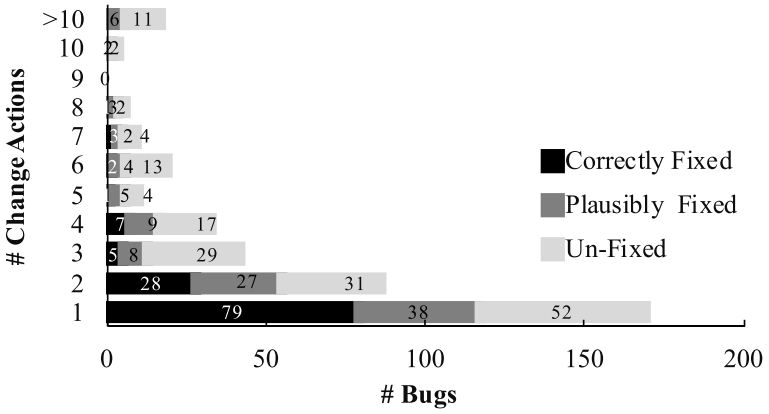


Fig. 11 Repairability of APR systems with respect to the number of change actions

need more change actions. For example, only 7 (=1+2+3+1) out of 55 bugs that need at least five change actions, can be correctly fixed by at least one of the 24 state-of-the-art APR systems. In the literature, only Hercules authors claim that it can fix multi-hunk bugs (Saha et al. 2019) but only 4 bugs that need more than two change actions can be correctly fixed by Hercules. The only bug that was fixed by 8 change actions is correctly fixed by LSRepair (Liu et al. 2018). However, LSRepair fixes bugs by updating the buggy code at the method level but not at the statement level as change actions.

Table 3 provides a detailed view of the repairability performance of the different tools based on the number of change actions.

Interestingly, we find that 5 systems (i.e., ACS, VFix, SimFix, Hercules, and LSRepair) have the ability to generate correct patches for more "complex" bugs in terms of the number of change actions. We further assess the corresponding patches generated by these APR systems (except for VFix which does not come with a public artefact of generated patches). We found several reasons why the enumerated tools can fix complex bugs:

1. *Repeated change actions and adequate test cases*: The Time-26 bug that was correctly patched by Hercules consists of 7 change actions but they can be classified into two unique change actions (i.e., inserting a parameter of a method invocation in the `ReturnStatement` and inserting a parameter of a method invocation in the `VariableDeclarationStatement`) with the complete change actions from its developer patch. In addition, the associated test suite includes 8 independent failed test cases. Both make it possible for Hercules to repair this bug hunk by hunk.
2. *Apply fewer change actions than those of ground truth (i.e., human-written patches) to generate semantically equivalent patches*: Lang-41 and Lang-50 contain 7 and 6 change actions respectively, but SimFix applies 2 change actions to each one during the bug fixing. Also, ACS employs 2 change actions for fixing Math-93 whose ground truth patch is comprised of 5 change actions.
3. *Method level change actions*: LSRepair fixes Lang-46 consisting of 8 change actions by performing change operation at the method level.

Figure 12 provides statistics on the relationship between code change actions depth with repair performance.

Table 3 With respect to the number of change actions that each APR system could implement in a given patch

# change actions	APR systems
1	All 24 APR Tools
2	16 (GENPAT, ConFix, Vfix, PraPR, TBar, kPAR, ARJA, Hercules, FixMiner, LSRepair, SimFix, ssFix, ACS, HDRepair, Nopol, AVATAR)
3	9 (GENPAT, Vfix, PraPR, TBar, Hercules, SimFix, JAID, ssFix, AVATAR)
4	9 (ConFix, Vfix, TBar, ARJA, Hercules, LSRepair, CapGen, ACS, AVATAR)
5	ACS
6	Vfix, SimFix
7	SimFix, Hercules
8	LSRepair

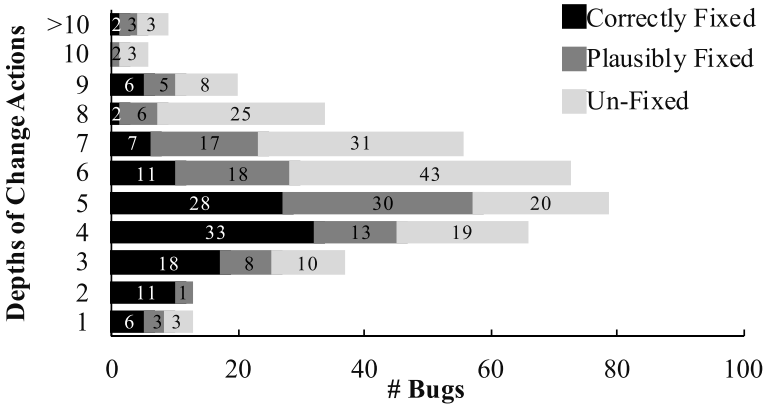


Fig. 12 Relationship between the fix/unfixed bugs and the depths of change actions

The bugs that are correctly fixed by existing APR systems have relatively low depth of code change actions. For example, ∼77% of correctly fixed bugs can be addressed with change actions whose depths are lower or equal to 5. Nevertheless, APR systems manage to generate deeper changes (i.e., that are invasive in the AST tree) that are correct. The example fix generated by SimFix and illustrated in Fig. 13 puts into perspective the common criticism that APR fixes simple bugs.

Measurement of complexity in terms of change actions breadth and depth shows that many correctly fixed bugs by literature tools are indeed among the least complex. Nevertheless, the statistics also show the progress achieved by state-of-the-art APR systems on repairing complex bugs since some correct patches are relatively complex.

►**RQ-3.2**◄ *How does the locality of donor code for Defects4J bugs impact the performance of APR systems?*

Figure 14 overviews the statistics about relationship between the donor code locality and fixed/unfixed bugs. “Same” and “Different” have the same meaning as they are in Fig. 10, respectively.

```
--- org/apache/commons/lang/math/NumberUtils.java
+++ org/apache/commons/lang/math/NumberUtils.java
@@ -452,4 +452,4 @@ public static Number createNumber
     if (dec == null
         && exp == null
-        && isDigits(numeric.substring(1))
-        && (numeric.charAt(0) == '-' || Character.isDigit(numeric.charAt(0)))) {
+        && (numeric.charAt(0) == '-' && isDigits(numeric.substring(1)) ||
     isDigits(numeric))) {
```

Fig. 13 Example of bugs fixed by complex change actions (Lang-58 in Defecst4J)

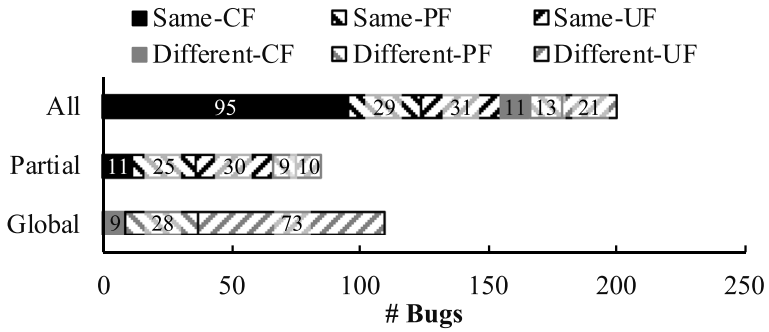


Fig. 14 Relationship between donor code locality and fixed/unfixed bugs. CF, PF, UF represent Correctly Fixed, Plausibly Fixed, and Un-Fixed, respectively

As shown in Fig. 14, the donor code of most ($\sim$84%=$\frac{95+11}{95+11+11+9}$) correctly fixed bugs is available from the buggy program. The existing APR systems still can correctly fix 11 and 9 bugs even their donor code is partially available or unavailable from the buggy program, respectively. Nevertheless, there is a large space to improve their repair performance. For example, there are 94(=29+31+13+21) bugs cannot be correctly fixed by any APR tool yet, even though all necessary donor code for fixing them are available from the buggy program. Additionally, $\sim$44% =($\frac{25+30+9+10+28+73}{395}$) of bugs that cannot be correctly fixed simply because the necessary donor code is not fully available from the buggy program. Therefore, for the future APR work, practitioners should pay more attention to the bugs of which donor code is partially available or even unavailable from the buggy programs themselves.

Most APR systems mainly focus on addressing bugs whose donor code is available from the buggy program. Considering code outside the buggy program can improve the quality of patch candidates

Feasibility of Mutating Intrinsic Donor Code As presented in Fig. 15, most intrinsic donor code required by bugs can be correctly fixed by the existing APR systems. There are still 5 plausibly-fixed and 7 un-fixed bugs that cannot be fixed even their necessary donor code is intrinsic. When donor code are available from a larger search space, APR systems tend to be unable to fix bugs. For example, if it is necessary to scan at least local packages for donor code, APR systems can correctly fix only 8 bugs while no system can fix 33 bugs even with the search space of local packages.

It is feasible to fix bugs by directly mutating the intrinsic donor code, but there still is a space of improving the repair performance to boost APR by addressing intrinsic code related bugs.

Trade-off between search space and efficiency As presented in Fig. 9, it is able to find all necessary donor code for more bug fixes when enlarging the search space from the local buggy method to the whole buggy program. However, enlarging the search space of donor code will sharply enlarge the search space of patch candidates that will further increase the possibility of generating plausible but incorrect patches before the correct ones (Wen et al.

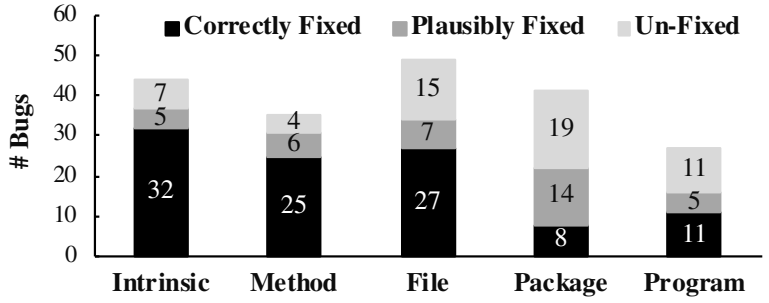


Fig. 15 Relationship between the fixed/unfixed bugs and the donor code categories

2018). When enlarging the search space from the buggy file to the whole buggy program, 60% ($=\frac{25+16}{25+16+16+11}$) of the increased bugs, of which donor code are available from the program, are served with the type-2 donor code. Nevertheless, the number of patch candidates generated with type-2 donor code will be increased exponentially as the number of potential replaceable identifiers in buggy code.

From the aspect of repairability, other 19 (=8+11) bugs can be fixed by existing APR tools, but the other 19 (=14+5) bugs can be fixed in a plausible but incorrect way. Enlarging the search space leads to higher cost for APR, but it can impact its efficiency of APR system by reducing the precision of generated correct patches. The trade-off between fixing more bugs and correctly fixing more bugs should be taken into account for the selection of donor code searching space.

When considering a large donor code search space, it is possible to fix more bugs for APR systems, but the trade-off (Long and Rinard 2016) *between the repair performance in terms of the number of fixing bugs and the efficiency in terms of the time cost or patch precision should be carefully maintained by practitioners to boost APR.*

6 Discussion

6.1 Considering All Commits for Mining Change Actions

In the APR community, bug-fixing related keywords (e.g., "bug", "error", "fault", "fix", "patch" and "repair") have been widely used to explore the characteristics of human patches (Liu et al. 2018; Nguyen et al. 2013; Barr et al. 2014; Martinez et al. 2014; Zhong and Su 2015) or mine fix patterns from human patches (Kim et al. 2013; Le et al. 2016; Jiang et al. 2018; Wen et al. 2018). However, some bug-fixing commits might not be collected by the keyword searching because of developers inconsistent convention of committing messages (Zhong and Su 2015). And it is possible to find the change actions from non-bug fixing commits for fixing bugs. Intuitively, more bugs could be matched with adequate change actions if considering all historical commits. Therefore, we perform a parallel experiment considering all commits in the buggy program repository on 395 bugs in the Defects4J dataset to match change actions for each of the 395 bugs. In the parallel analysis, we consider three statistics of change actions: (1) availability: how many bugs can be matched with adequate change actions, (2) search space: how many historical commits are considered to

find the adequate change actions for fixing bugs, and (3) accessibility: how the matched change actions are ranked with their frequencies. The related results are shown in Table 4.

Indeed, more bugs can be matched with adequate change actions when considering a big search space of commits (i.e., all commits against bug-fixing related commits). However, the search space of commits is increased sharply over three times (897/292=3.07X). It will enlarge the costs on mining adequate change actions for fixing bugs if considering all commits. Furthermore, the average of frequency ranks is increased from 140 to 324. It means that the performance of fixing bugs with change actions by considering all commits will be impacted in two aspects: ❶ the increased search space of change actions will reduce the efficiency of finding the adequate change actions for fixing bugs, and ❷ the adequate change actions are not ranked prioritizedly, which makes more trials with inadequate change actions (1) to generate more non-sensical patch candidates to reduce the efficiency of fixing bugs and (2) to increase the possibility of generating plausible patches with the inadequate change actions to reduce the precision of generated patches.

To sum up, considering all commits for collecting change actions can make more bugs be matched with adequate change actions than only considering bug-fixing related commits. However, the efficiency and precision of fixing bugs will be impacted negatively. As stated in recent studies, efficiency (Liu et al. 2020; Qin et al. 2021) and precision (Xiong et al. 2017; Xiong et al. 2018; Le et al. 2019; Tian et al. 2020) have become the two important metrics of evaluating the bug-fixing performance of APR systems. Therefore, it would be reasonable to consider the bug-fixing related commits to collect/mine change actions for program repair. The availability of change actions increases when including all commits. However, the search space becomes much larger and the accessibility decreases sharply, which makes it more difficult to find adequate change actions for fixing a bug.

6.2 Threats to Validity

External validity Our study only considers the Defects4J dataset. All findings might thus be valid only for this benchmark. Nevertheless, this threat is mitigated by the fact that all bugs and their patches are collected from six real-world programs. The curated dataset is the most widely used dataset in the APR community, where the isolated bugs and fixes provide the ground truth for APR study. Our study relies on the revision history of a buggy program, it will be a threat when the related revision history is unavailable.

Internal validity Our implementation of collecting bug fix related commits as well as similar donor code searching may threaten the validity of some of our conclusions. We mitigate this threat by reusing common components of patch commit collection from the literature, and considering the first two types of code clone for similar donor code searching by

Table 4 Comparison of matching change actions for bugs with bug-fixing related commits with all commits

	Bug-fixing related commits	All commits
Number of matched bugs	117	152
Number of commits (on average)	292	897
Rank of change actions with frequency (on average)	140	324

referencing the donor code searching strategies in the APR tools. The internal threats include the donor code that is irrelevant to the context of buggy code but not covered by type-1 & 2. Note that type-3 & 4 clones are possible for fixing bugs, which is somehow explored by LSRepair (Liu et al. 2018). However, the type-3 & 4 clones are the state-of-the-art open research questions, it is challenging to evaluate the availability of type-3 & 4 clones for patch generation. Therefore, such two types of clones are not considered in this study.

Construct validity By construct, to reduce the bias from similar change actions, we only identify the availability of change actions only when the identical ones can be extracted from the history. However, this setting may penalize the identification of similar change actions that might be actionable. We mitigate this threat by selecting the identical identification, since the similar change actions could bias the exact repair performance of fix pattern based APR tools after a systematic review on the fix pattern studies in APR commit. In addition, the combination of change actions and donor code is necessary for patch generation. Our work investigates the possibility of obtaining the change actions and donor code from the buggy program with a wide view. The combination could narrow this view, thus it is not considered in this work.

7 Related Work

Bug Fixes In the literature, bug fixes have been widely studied to deepen knowledge for program repair. Pan et al. (2009), Martinez and Monperrus (2015), and Zhong and Su (2015) performed empirical investigations on Java program patches, mostly analyzing the repair actions of patches at the statement level. Liu et al. (2018) further propose a fine-grained study on 16,450 bug fix commits from seven Java open-source projects to deepen the knowledge. Sobreira et al. (2018) dissertate Defects4J bugs with repair actions to characterize bugs and patches. Jiang et al. (2019) investigate the difficulty of automated bug fixing on 50 Defects4J bugs through a manual analysis. Our study focuses on the distribution of repair ingredients for all Defects4J bug fixes and investigates the potential relationship between them and the repair performance of APR systems in a fully automated manner.

Change Actions Change actions (a.k.a., fix patterns, fix templates, code transformations etc.) are also broadly studied to boost automated program repair. Weimer et al. (2009) and Goues et al. (2012) rely on simple change actions of two genetic programming operators to propose the milestone GenProg. Kim et al. (2013) manually summarize 10 fix templates from 62,656 human-written patches for program repair. Long et al. (2017) propose to automatically learn code transformations from real-world patches. Yue et al. (2017) manually obtained 16 fix patterns from 19,275 fixing patches from three real-world projects. Liu et al. (2018) leverage the deep learning techniques to extract fix patterns from FindBugs violations (Hovemeyer and Pugh 2004). Liu and Zhong (2018) mine fix patterns from the public forum of StackOverflow. Wen et al. (2018) and Jiang et al. (2018) also leverage the change actions extracted from a public dataset of bug fixes (Le et al. 2016) to build their APR systems. However, none of existing APR systems is proposed to leverage the change actions extracted from the buggy program itself to proceed the patch generation. Our work proposes to investigate the possibility of collecting change actions from the buggy programs themselves.

Nguyen et al. (2013) compared the within-project repetitiveness of change actions with the cross-project one and concluded that the former is generally lower than the latter.

However, the study is limited by splitting the change actions of a bug fix into multiple change actions of different sizes (i.e., the tree height defined in Nguyen et al. (2013)). Thus, the study fails to demonstrate the exact distributions of raw change actions consisting of a bug fix. Moreover, their work did not explore the reasons for the low availability of change actions inside the buggy program history. Our work investigates the "available", "partially available", and "unavailable" situations for each bug fix in terms of change actions. In addition, we explore the underlying reasons of the low availability of change actions and further measure the impact of change actions on the repair ability of 24 APR systems.

Donor Code The APR community investigates donor code as one of the essential repair ingredients for program repair. Barr et al. (2014) explored more than 15K commits in 12 open-source projects and discovered that 43% of code changes are graftable. This study is, however, limited by the line-level granularity. Martinez et al. studied the redundancy assumption of repair ingredients at line and token granularities (Martinez et al. 2014) but the study presents only temporal-redundancy (i.e., how many commits contain redundant changes) rather than spatial redundancy (i.e., where are they?). PAR (Kim et al. 2013) limits the donor code search in the code files that contain the bugs, which is followed by other state-of-the-art APR systems (e.g., kPAR (Liu et al. 2019), TBar (Liu et al. 2019a) and Avatar (Liu et al. 2019b)). CapGen (Wen et al. 2018) and SimFix (Jiang et al. 2018) perform the fine-grained donor code search from the whole buggy program. ACS (Xiong et al. 2017) is proposed to extract donor code from the bug-triggering test cases with specific constraints. LSRepair (Liu et al. 2018) even conducts a live search of donor code from real-world programs. Our work aims at assessing to what extent donor code is distributed in the buggy program to boost program repair.

8 Conclusion

This paper reports on the result of a systematic study on the need for repair ingredients (i.e., change actions and donor code) for bug fixes. It is challenging, on the one hand, to extract all necessary change actions from the buggy program to fix its bug, when the program does not have enough commits related to bug fixes. On the other hand, donor code for most bugs is available from the buggy program, but the trade-off between the search space of donor code and the repair efficiency of APR systems should be non-trivially maintained by practitioners. To boost program repair, it is necessary to probe advanced systems to address difficult bugs with respect to the complicated code change actions, as well as advanced searching strategies to efficiently find donor code for patch synthesizing.

Acknowledgements This research was supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06) and the Open Project Program of the Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology (No. XCA20026), the National Key R&D Program of China (No. 2020AAA0107704), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1A5A1021944), the National Defense Basic Scientific Research Program (No. WDZC20205500308), the Fundamental Research Funds for the Central Universities (Nos. 2021CDJKYJH032, 2020CDCGRJ037, 2020CDCGRJ072) and the National Natural Science Foundation of China (Nos. 61872445, 61672529), as well as the funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949014).

Data Availability All relevant artifacts of our study are publicly available at: https://github.com/DehengYang/repair-ingredients.

References

Barr ET, Brun Y, Devanbu PT, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering. ACM, pp 306–317

Britton T, Jeng L, Carver G, Cheak P, Katzenellenbogen T (2013) Reversible debugging software. Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep

Chen L, Pei Y, Furia CA (2017) Contract-based program repair without the contracts. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE, pp 637–647

Durieux T, Cornu B, Seinturier L, Monperrus M (2017) Dynamic patch generation for null pointer exceptions using metaprogramming. In: Proceedings of the 24th international conference on software analysis, evolution and reengineering. IEEE, pp 349–358

Durieux T, Madeiral F, Martinez M, Abreu R (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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp 302–313

Falleri J-R, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International conference on automated software engineering. ACM, pp 313–324

Gazzola L, Micucci D, Mariani L (2017) Automatic software repair: A survey. IEEE Trans Softw Eng 45(1):34–67

Ghanbari A, Benton S, Zhang L (2019) Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT International symposium on software testing and analysis. ACM, pp 19–30

GitHub (2021) Defects4j . https://github.com/rjust/defects4j/releases/tag/v1.4.0

Goues CL, Dewey-Vogt M, Forrest S, Weimer W (2012) A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International conference on software engineering. IEEE, pp 3–13

Goues CL, Nguyen TV, Forrest S, Weimer W (2012) GenProg: A generic method for automatic software repair. IEEE Trans Softw Eng 38(1):54–72

Goues CL, Pradel M, Roychoudhury A (2019) Automated program repair. Commun ACM 62(12):56–65

Gupta R, Pal S, Kanade A, Shevade S (2017) Deepfix: Fixing common c language errors by deep learning. In: Proceedings of the 31st AAAI conference on artificial intelligence. AAAI, pp 1345–1351

Hovemeyer D, Pugh W (2004) Finding bugs is easy. ACM sigplan notices 39(12):92–106

Hua J, Zhang M, Wang K, Khurshid S (2018) Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th international conference on software engineering. ACM, pp 12–23

Jiang L, Misherghi G, Su Z, Glondum S (2007) Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International conference on software engineering (ICSE'07). IEEE, pp 96–105

Jiang J, Ren L, Xiong Y, Zhang L (2019) Inferring program transformations from singular examples via big code. In: Proceedings of the 34th IEEE/ACM International conference on automated software engineering. pp 255–266

Jiang J, Xiong Y, Xia X (2019) A manual inspection of defects4j bugs and its implications for automatic program repair. Sci Chin Inf Sci 62(10):200102

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (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. ACM, pp 298–309

Just R, Jalali D, Ernst MD (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. ACM, pp 437–440

Just R, Parnin C, Drosos I, Ernst MD (2018) Comparing developer-provided to user-provided tests for fault localization and automated program repair. In: Proceedings of the 27th ACM SIGSOFT International symposium on software testing and analysis. ACM, pp 287–297

Kim J, Kim S (2019) Automatic patch generation with context-based change application. Empir Softw Eng 24(6):4071–4106

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 35th International conference on software engineering. IEEE, pp 802–811
Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Traon YL (2019) Fixminer: Mining relevant fix patterns for automated program repair. Empir Softw Eng 25(3):1980–2024
Koyuncu A, Liu K, Bissyandé TF, Kim D, Monperrus M, Klein J, Traon YL (2019) iFixR: Bug report driven program repair. In: Proceedings of the 27the ACM joint european software engineering conference and symposium on the foundations of software engineering. ACM, pp 314–325
Le X-BD, Bao L, Lo D, Xia X, Li S, Pasareanu C (2019) On reliability of patch correctness assessment. In: Proceedings of the 41st International conference on software engineering. IEEE, pp 524–535
Le XBD, Lo D, Goues CL (2016) History driven program repair. In: Proceedings of the 23rd IEEE International conference on software analysis, evolution, and reengineering. IEEE, pp 213–224
Liu K, Kim D, Bissyandé TF, Yoo S, Traon YL (2018) Mining fix patterns for findbugs violations. IEEE Trans Softw Eng 47(1):165–188
Liu K, Kim D, Koyuncu A, Li L, Bissyandé TF, Traon YL (2018) A closer look at real-world patches. In: 2018 IEEE International conference on software maintenance and evolution (ICSME). IEEE, pp 275–286
Liu K, Koyuncu A, Bissyandé TF, Kim D, Klein J, Traon YL (2019) You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: Proceedings of the 12th IEEE International conference on software testing, verification and validation. IEEE, pp 102–113
Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) TBar: Revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis. ACM, pp 31–42
Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: Proceedings of the 26th IEEE international conference on software analysis, evolution and reengineering. IEEE, pp 456–467
Liu K, Koyuncu A, Kim K, Kim D, Bissyandé TF (2018) LSRepair: Live search of fix ingredients for automated program repair. In: Proceedings of the 25th Asia-Pacific Software Engineering Conference. IEEE, pp 658–662
Liu K, Li Li, Koyuncu A, Kim D, Liu Z, Klein J, Bissyandé TF (2021) A critical review on the evaluation of automated program repair systems. J Syst Softw 171:110817
Liu K, Wang S, Koyuncu A, Kim K, Bissyandé TF, Kim D, Wu P, Klein J, Mao X, Traon YL (2020) On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: Rothermel G, Bae D-H (eds) Proceedings of the 42nd International conference on software engineering. ACM, pp 615–627
Liu X, Zhong H (2018) Mining stackoverflow for program repair. In: Proceedings of the 25th IEEE international conference on software analysis, evolution and reengineering. IEEE, pp 118–129
Long F, Amidon P, Rinard M (2017) Automatic inference of code transforms for patch generation. In: Proceedings of the 11th joint meeting on foundations of software engineering. ACM, pp 727–739
Long F, Rinard M (2016) An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International conference on software engineering. IEEE, pp 702–713
Lou Y, Chen J, Zhang L, Hao D, Zhang L (2019) History-driven build failure fixing: how far are we? In: Proceedings of the 28th ACM SIGSOFT International symposium on software testing and analysis. ACM, pp 43–54
Martinez M, Durieux T, Sommerard R, Xuan J, Monperrus M (2017) Automatic repair of real bugs in java A large-scale experiment on the defects4j dataset. Empir Softw Eng 22(4):1936–1964
Martinez M, Martin Monperrus. (2016) Astor: A program repair library for java. In: Proceedings of the 25th International symposium on software testing and analysis. ACM, pp 441–444
Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng 20(1):176–205
Martinez M, Weimer W, Monperrus M (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. ACM, pp 492–495
Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the international conference on software maintenance. pp 120–130
Monperrus M (2018) Automatic software repair: A bibliography. ACM Comput Surv 51(1):17:1–17:24
Motwani M, Sankaranarayanan S, Just R, Brun Y (2018) Do automated program repair techniques repair hard and important bugs? Empir Softw Eng 23(5):2901–2947

Nguyen HA, Nguyen AT, Nguyen TT, Nguyen TN, Rajan H (2013) A study of repetitiveness of code changes in software evolution. In: 2013 28th IEEE/ACM International conference on automated software engineering (ASE), pp 180–190

Pan K, Kim S, James Whitehead E (2009) Toward an understanding of bug fix patterns. Empir Softw Eng 14(3):286–315

Qi Z, Long F, Achour S, Rinard M (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the international symposium on software testing and analysis. ACM, pp 24–36

Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp 254–265

Qi Y, Mao X, Lei Y, Wang C (2013) Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the 22nd International symposium on software testing and analysis. ACM, pp 191–201

Qi X, Reiss SP (2017) Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering. IEEE, pp 660–670

Qin Y, Wang S, Liu K, Mao X, Bissyandé TF (2021) On the impact of flaky tests in automated program repair. In: Proceedings of the 28th IEEE International conference on software analysis, evolution and reengineering. IEEE, pp 295–306

Rolim R, Soares G, Gheyi R, D'Antoni L (2018) Learning quick fixes from code repositories. arXiv preprint arXiv:1803.03806

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci Comput Programm 74(7):470–495

Saha RK, Lyu Y, Yoshida H, Prasad MR (2017) Elixir: Effective object-oriented program repair. In: Proceedings of the 32nd IEEE/ACM International conference on automated software engineering. IEEE, pp 648–659

Saha S, Saha RK, Prasad MR (2019) Harnessing evolution for multi-hunk program repair. In: Proceedings of the 41st international conference on software engineering. IEEE, pp 13–24

Sobreira V, Durieux T, Madeiral F, Monperrus M, de Almeida Maia M (2018) Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In: Proceedings of the IEEE 25th International conference on software analysis, evolution and reengineering. IEEE, pp 130–140

Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF (2020) Evaluating representation learning of code changes for predicting patch correctness in program repair. In: Proceedings of the 35th IEEE/ACM International conference on automated software engineering. IEEE, pp 981–992

Weimer W, Nguyen TV, Goues CL, Forrest S (2009) Automatically finding patches using genetic programming. In: Proceedings of the 31st international conference on software engineering. IEEE, pp 364–374

Wen M, Chen J, Wu R, Hao D, Cheung S-C (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International conference on software engineering. ACM, pp 1–11

Xiong Y, Liu X, Zeng M, Zhang L, Huang G (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International conference on software engineering. ACM, pp 789–799

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise condition synthesis for program repair. In: Proceedings of the 39th IEEE/ACM international conference on software engineering. IEEE, pp 416–426

Xu X, Sui Y, Yan H, Xue J (2019) Vfix: value-flow-guided precise program repair for null pointer dereferences. In: Proceedings of the 41st International Conference on Software Engineering. IEEE, pp 512–523

Xuan J, Martinez M, Demarco F, Clement M, Marcote SL, Durieux T, Le Berre D, Monperrus M (2017) Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans Softw Eng 43(1):34–55

Xuan J, Monperrus M (2014) Test case purification for improving fault localization. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering. ACM, pp 52–63

Yang D, Qi Y, Mao X (2017) An empirical study on the usage of fault localization in automated program repair. In: Proceedings of the 33rd IEEE International conference on software maintenance and evolution. IEEE, pp 504–508

Yang D, Qi Y, Mao X, Lei Y (2021) Evaluating the usage of fault localization in automated program repair: an empirical study. Front Comput Sci 15(1):1–15

Yuan Y, Banzhaf W (2018) Arja: Automated repair of java programs via multi-objective genetic programming. IEEE Trans Softw Eng

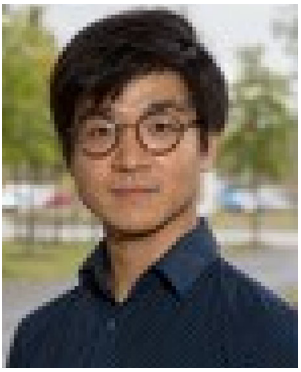
Yue R, Na M, Wang Q (2017) A characterization study of repeated bug fixes. In: 2017 IEEE International conference on software maintenance and evolution (ICSME). IEEE, pp 422–432
Zhong H, Su Z (2015) An empirical study on real bug fixes. In: Proceedings of the 37th international conference on software engineering. IEEE, pp 913–923

Deheng Yang is currently a Ph.D. student at National University of Defense Technology, under the supervision of Dr. Xiaoguang Mao. He received the BA in computer science and technology from the National University of Defense Technology. His research interests include fault localization, automated program repair, etc.

Kui Liu is an associate professor in Software Engineering at the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China. He received the MS degree in computer application technology at Southwest University (Chongqing, China) in 2013, and obtained the Ph.D. degree in computer science at the University of Luxembourg in 2019. His research interests include automated program repair, automated fault localization, fix pattern mining, deep learning, and empirical software engineering.

Dongsun Kim is Assistant Professor of the School of Computer Science at Kyungpook National University. He was formerly software engineer at Furiosa.ai, research associate at the University of Luxembourg, and post-doctoral fellow at the Hong Kong University of Science and Technology. His research interest includes testing AI systems, automatic patch generation, fault localization, static analysis, and search-based software engineering. In particular, automated debugging is his current focus. His recent work has been recognized by several awards, such as a featured article of the IEEE Transactions on Software Engineering (TSE) and ACM SIGSOFT Distinguished Paper of the International Conference on Software Engineering (ICSE). He has led the FIXPATTERN project funded by FNR (Luxembourg National Research Fund) CORE programme. He is now leading the PreDebugging project funded by NRF (National Research Foundation of Korea) Regional Researcher Program.

Anil Koyuncu is an assistant professor at the Computer Science and Engineering Program of Faculty of Engineering and Natural Sciences at Sabanci University, Turkey. He received his MS degree at Politecnico di Milano and his PhD degree from the University of Luxembourg. His research interests lie in the general area of software engineering, with particular emphasis on automated program repair. His work draws on data mining, program analysis and fault localization and aims to improve the automated program repair agenda towards boosting its adaption by practitioners.

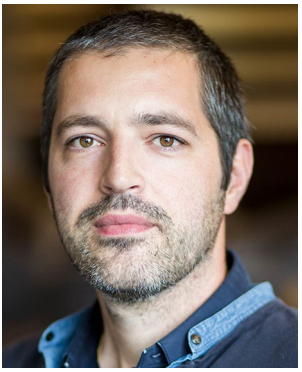


Kisub Kim is PhD student at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) in the University of Luxembourg. He received his master degree in computer engineering from Chungbuk National University, Cheongju, Korea, in 2017. His research interests include source code search, mining software repositories, and requirement engineering.

Haoye Tian is PhD student at the Interdisciplinary Centre for Security, Reliability and Trust (SnT) in the University of Luxembourg. He received his master degree in Computer Science from the Chongqing University, China in 2019. His research interests include automated program repair, patch validation, deep learning.

Yan Lei received the BA, MA and Ph.D. degrees in computer science and technology, all from the National University of Defense Technology, China. He is an associate professor at the School of Big Data & Software Engineering in Chongqing University, China. His research interests include fault localization, program repair, program slicing, etc.

Xiaoguang Mao is a professor at College of Computer, National University of Defense Technology, China. His research interests include high confidence software, software development methodology, software assurance, software service engineering, etc.



Jacques Klein is a researcher and professor in software engineering and software security who develops innovative approaches and tools towards helping the research and practice communities build trustworthy software. He is a member of the Interdisciplinary Centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. He received a Ph.D. degree in Computer Science from the University of Rennes, France, in 2006. His main areas of expertise are threefold: (1) Software Security (Malware detection, prevention and dissection, Static Analysis for Security, Vulnerability Detection, etc.); (2) Software Reliability (Software Testing, Semi-Automated and Fully-Automated Program Repair, etc.); (3) Data Analytics (Multi-objective reasoning and optimization, Model-driven data analytic, Time Series Pattern Recognition, etc.).

Tegawendé F. Bissyandé is research scientist with the Interdisciplinary Center for Security, Reliability and Trust (SnT) at the University of Luxembourg. He holds a PhD in computer from the Université de Bordeaux in 2013, and an engineering degree (MSc) from ENSEIRB. His research interests are in debugging, including bug localization and program repair, as well as code search, including code clone detection and code classification.

Affiliations

Deheng Yang[1] · Kui Liu[2,3,4] · Dongsun Kim[5] · Anil Koyuncu[6] · Kisub Kim[7] · Haoye Tian[7] · Yan Lei[8] · Xiaoguang Mao[1] · Jacques Klein[7] · Tegawendé F. Bissyandé[7]

Deheng Yang
yangdeheng13@nudt.edu.cn

Dongsun Kim
darkrsw@knu.ac.kr

Anil Koyuncu
anil.koyuncn@sabanciuniv.edu

Kisub Kim
kisub.kim@uni.lu

Haoye Tian
haoye.tian@uni.lu

Yan Lei
yanlei@cqu.edu.cn

Xiaoguang Mao
xgmao@nudt.edu.cn

Jacques Klein
jacques.klein@uni.lu

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

1 National University of Defense Technology, Changsha, China

2 Nanjing University of Aeronautics and Astronautics, Nanjing, China

3 State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China

4 Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, China

5 Kyungpook National University, Daegu, South Korea

6 Sabanci University, Istanbul, Turkey

7 SnT, University of Luxembourg, Luxembourg City, Luxembourg

8 Chongqing University, Chongqing, China