Noname manuscript No.
(will be inserted by the editor)

Towards Reusing Hints from Past Fixes

An Exploratory Study on Thousands of Real Samples

Hao Zhong · Na Meng

Received: date / Accepted: date

Abstract With the usage of version control systems, many bug fixes have accumulated over the years. Researchers have proposed various automatic program repair (APR) approaches that reuse past fixes to fix new bugs. However, some fundamental questions, such as how new fixes overlap with old fixes, have not been investigated. Intuitively, the overlap between old and new fixes decides how APR approaches can construct new fixes with old ones. Based on this intuition, we systematically designed six overlap metrics, and performed an empirical study on 5,735 bug fixes to investigate the usefulness of past fixes when composing new fixes. For each bug fix, we created *delta dependency graphs* (*i.e.*, program dependency graphs for code changes), and identified how bug fixes overlap with each other in terms of *the content, code structure, and identifier names of fixes*. Our results show that if an APR approach composes new fixes by fully or partially reusing the content of past fixes, only 2.1% and 3.2% new fixes can be created from single or multiple past fixes in the same project, compared with 0.9% and 1.2% fixes created from past fixes across projects. However, if an APR approach composes new fixes by fully or partially reusing the code structure of past fixes, up to 41.3% and 29.7% new fixes can be created. By making the above observations and revealing other ten findings, we investigated the upper bound of reusable past fixes and composable new fixes, exploring the potential of existing and future APR approaches.

This paper is an extended version of a poster paper [50] that is presented in the 39th International Conference on Software Engineering, 2017.

H. Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
E-mail: zhonghao@sjtu.edu.cn

N. Meng
Department of Computer Science
Virginia Tech, USA
E-mail: nm8247@cs.vt.edu

1 Introduction

With the usage of version control systems, many bug fixes have accumulated over the years. Researchers conducted various empirical studies to understand bug fixes [8, 14, 30–32, 35]. For example, Nguyen *et al.* found that bugs can be repetitive [32], indicating that it is feasible to fix new bugs using past fixes. Based on such observations, other researchers proposed some automatic program repair (APR) approaches that reuse past fixes to fix new bugs. For example, Kim *et al.* extracted fix patterns from thousands of bug fixes [13]. Martinez and Monperrus mined repair models from past bug fixes to guide the repair process [25]. Long *et al.* trained a model with past fixes to automatically generate patches for repairing incorrect applications [23]. Although these approaches show promising results, they are limited in the following two aspects:

1. Existing empirical studies are based on manual or simple automatic analysis. To understand the overlap between bug fixes, researchers usually first check out bug-fixing revisions in software version histories, and then identify source files modified for bug fixes. Some researchers manually compared fixes for recurring bug-fixing patterns [13, 38]. However, this process is tedious, error-prone, and subject to human bias. Some other researchers analyzed bug fixes [25,48,51] leveraging tools like ChangeDistiller [4], PPA [3], and CCFinder [12]. For instance, Martinez *et al.* used ChangeDistiller to represent changes as edit scripts to the abstract syntax trees (ASTs) of modified source files, and then extracted common edit operations [25]. Zhong *et al.* used PPA to perform type inference on partial Java programs [51]. Yue *et al.* used CCFinder to compare the textual diff regions in modified files for repeated fixes [48]. However, none of these tools is able to correlate changes applied in different software entities (i.e., classes, methods, and fields) based on their data- or control- dependence relationships; neither can existing studies investigate the more sophisticated bug fixing patterns that involve multiple software entities.

2. Some hypotheses of APR approaches are not fully explored. For instance, Kim *et al.* [13] and Long *et al.* [23] generated new fixes from past fixes based on the following hypothesis: the successful human patches applied in the version history of different software share certain characteristics. This hypothesis is built on previous studies by Barr *et al.* [1] and Martinez *et al.* [26]. In particular, Barr *et al.* [1] reported that 11% bug fixes could be fully reconstituted from existing code, and Martinez *et al.* [26] observed that 3% to 17% bug fixes are temporally redundant, when they split bug fixes into code lines. However, the two studies may over- or under- estimate the graftability or redundancy of bug fixes for two reasons. First, when splitting code changes into snippets, and match snippets between new fixes and existing code or fixes purely based on similar text, these studies lose the syntactic structures of changes, and any dependency relationship or identifier usage between relevant changes. As a result, they overestimated the reusability of past fixes, because it is very unlikely that any APR approach will generate patches by enumerating all possible combinations of code lines in existing codebases or applied fixes. Second, when fixes are matched purely based on the textual similarity, these studies can miss textually dissimilar, but syntactically similar fixes, underestimating the reusability of past fixes. Therefore, it is still unknown how bug fixes overlap in terms of code structures and identifier

names of classes, methods, and fields. Although the current hypothesis is correct in some cases, we do not know how well it generalizes.

In this study, we aim to further investigate the reusable hints from past fixes. Our study does not suffer from the above two problems, since it does not split bug fixes into snippets before comparison, and it introduces more advanced code analysis for comparison. The benefits of our study are as follows:

Benefit 1. The answers can deepen our knowledge on bugs and how to fix bugs effectively. For example, while many programmers believe that it is quite difficult to fix bugs, some recent studies [32] show contradicted evidences. With the answers, we can reduce such controversies.

Benefit 2. The answers can provide insights on the potential of this research area, which can reduce superficial expectations and motivate repair approaches that fulfil the potential.

Since some existing APR approaches construct fixes from past fixes [13, 23], we performed an empirical study to investigate the usefulness of past fixes by measuring the overlap between past fixes and current fixes. Hypothetically, the more overlap a current fix has with past fixes, the more likely that it can be constructed from those fixes. To compute the overlap, we have to overcome the following challenges:

Challenge 1. Bug fixes are difficult to understand. As a bug fix may involve modifications to multiple methods, many prior empirical studies are done via manual code inspection [10, 24, 47]. However, such manual checking process is not scalable, and may suffer from human bias. To make our study objective and comprehensive, we need to automate the process.

Challenge 2. Existing program comprehension tools are not sufficient. Although static analysis frameworks (*e.g.*, WALA[1] and Soot[2]) can conduct inter-procedural analysis to correlate code in different methods, they require for a whole program—including all source code and every library on which the program depends—to reason about the data- and control- dependency relationships inter-procedurally. To analyze such dependency relationships between code changes for each revision committed by developers in a timely manner, we cannot afford the exorbitant analysis time cost of applying whole-program analysis to both the before- and after- version of the revision, especially when most of the time is spent in resolving library dependencies and analyzing unchanged code. Partial Program Analysis (PPA) is a framework that performs partial type inference and uses heuristics to recover the declared type of expressions and resolve ambiguities in partial Java programs [3, 31]). However, it does not support inter-procedural program analysis to correlate changes, neither does it detect code changes. ChangeDistiller [4] and Unix diff [29] detect code changes without identifying the dependency relationships between changes.

To overcome the above challenges, we leverage our prior work, GRAPA [50], which combines ChangeDistiller, WALA, and PPA, to perform inter-procedural dependency analysis on code changes, and to create *delta dependency graphs (DDGs)* for related changes in different methods. By comparing each fix's DDGs produced by GRAPA, we performed a comprehensive empirical study on 5,735 bug fixes in

[1] http://wala.sourceforge.net

[2] https://github.com/Sable/soot

four popular projects: Aries, Cassandra, Derby, and Mahout. Our comparison checks for six types of overlap between bug fixes. *Namely, a bug fix may fully or partially overlap with one or multiple fixes in terms of code content, syntactic structures, and identifier names.* Imagining that all such overlap metrics can be leveraged by existing or future tools to automate patch generation, we explore how helpful past bug fixes can be when synthesizing new fixes. Our study provides the following insights:

- **Extent of constructing fixes from past fixes.** Bug fixes have more overlap in terms of syntactic structures than code content and identifier names (Findings 1 and 3). The syntactic structures of 41% fixes can be synthesized from the syntactic structures of multiple past fixes (Finding 2). A new fix is often relevant to only several useful past fixes (Findings 5 and 10). However, if we require exact matches, only 3% fixes can be constructed from past fixes (Finding 1), since, since most bugs need creative repairs that never appear in past fixes (Finding 4).
- **Preparing a repository of past fixes.** We consider a bug fix to be useful, if it is repetitive. If we build the repository from fixes of the same project, most fixes are useful (Finding 6), but it can be challenging to extract useful repair actions, since a fix typically has both useful repair actions and useless repair actions (Finding 7). However, the usefulness of a repository may not increase, with the increasing of collected bug fixes (Finding 10), and with more collected fixes, it is more difficult to identify useful ones (Finding 11). It is worthy identifying useful fixes, since an identified fix can be useful to repair many fixes (Finding 12).
- **Learning from other projects.** From other projects, it is feasible to learn as many code structure changes, but the combination of multiple past fixes do not achieve as good results as from fixes of the same projects. Furthermore, from other projects, as other projects typically have different client code names, it becomes more challenging to learn how to replace those API elements with correct ones (Findings 8 and 9).

Here, Application Programming Interface (API) code is a set of classes and methods provided by frameworks or libraries, and client code is application code that reuses or extends API classes and methods provided by API libraries. Our study does not draw any conclusion on the capability of existing patch generation tools. Instead, we try to explore the hypotheses underneath existing tools, such as how many useful hints can be learnt from past fixes, and how significant the divergence can be between current fixes and past fixes.

2 Open Questions

In this section, we break our research goal into the following research questions:

OP1:How many fixes can be fully constructed from past fixes?

Kim *et al.* [13] show that both code structure changes and name changes are useful to fix new bugs, and Long and Rinard [23] show that new fixes can be constructed from past fixes. However, it is unclear to what degree such fix patterns can be mined from past bug fixes. Here, we care about the potential, and assume that researchers can propose advanced approaches that reuse a change, even if its instance appears

only once in past bug fixes. Under this assumption, this research question investigates what can be learnt from past fixes, if we push mining techniques to their limits. Section 4.1 presents our answer to this question, which can be useful to design more effective mining techniques.

OP2: How creative is a bug fix?

Many programmers believe that it is difficult to fix bugs, since it takes even years to fix some bugs (*e.g.*, MySQL Bug `#20786`[3]). Although many researchers admit the complexity of fixing bugs, some recent studies present contradicted evidences. For example, Nguyen *et al.* [32] show that bug fixes can be repetitive, and Zhong and Su [51] show that many bug fixes do not need complicated repair actions. This research question mainly concerns the explanation for the contradicted evidences. It is related to the first question, but focuses on those changes that do not have overlaps. Here, we consider that a repair action is creative, if it does not appear in previous fixes. For each bug, we investigate how many of its nodes and methods can be covered by past fixes. If a change never appears in past fixes, it shall be more difficult to be fixed and needs more creative activities. In addition, if a bug needs to refer multiple past fixes, it shall be more difficult than those bugs that need to refer to only a past fix. For each bug, we investigate how many past fixes contribute to generating its current fix. Section 4.2 presents our answers to this question.

OP3. What are the challenges when preparing the repository of bug fixes?

For a bug under fixing, it needs to locate its related past fixes, before we can learn useful knowledge. For example, Long and Rinard [23] train a model to locate related bug fixes, before they use such fixes to guide the fixing process. This research question concerns how difficult it is to retrieve useful past fixes for a bug, which is reflected by the ratio from the useful past fixes to the total past fixes. In addition, for each past fix, we investigate how many fixes it can contribute and to what degree these fixes are useful. Section 4.3 presents our answers to this research question. For a mining technique, our results present the usefulness frequency of rocks, and can provide insights on designing effective mining techniques.

OP4. What is the potential to learn from other projects?

A project can have only limited past bug fixes, especially when the project is new. A natural way to handle this problem is to learn from other projects, but its effectiveness is largely unknown, since no previous approaches explored that and contradicted evidences are found in the related work. On one side, many approaches (*e.g.*, [53]) are proposed to mine specifications that define API usages across projects; on the other side, researchers find that some rules are specific to projects (*e.g.*, [20]). When investigating the previous three research questions, for each bug, we consider fixes on other bugs as past bug fixes, and we do not consider the commit time of bug fixes for simplicity. To investigate the fourth research question, we replace the past bug fixes with bug fixes from other projects, and redo the previous evaluations. Section 4.4 presents our answers to this research question.

[3] https://www.youtube.com/watch?v=oAiVsbXVP6k

3 Methodology

In this section, we present our dataset (Section 3.1), the underlying tool (Section 3.2), the major steps (Section 3.3), and our overlap metrics (Section 3.4). Like what Martinez *et al.* [26] did in their study, we compare a bug fix with known fixes to estimate whether it is feasible to construct the bug fix based on known bug fixes. However, our comparison is more detailed and accurate, since (1) our underlying tool compares SDGs for their deltas while their study compares textual contents and (2) we do not split bug fixes as they did. As we compare SDGs, code structure changes refer to any changes to edges, and code name changes refer to any changes to node labels. When comparing bug fixes, we envision three types of reusable components, such as code structure changes, code names changes, or their combinations, and we imagine two ways to leverage the reusable components: finding a matching component, or merging several components.

3.1 Dataset

We reuse the data set mentioned in prior work [51]. Each commit has a message. Based on messages, we define the following two criteria to identify bug fixes:
1. Issue number. All the projects in Table 1 have their issue trackers to track various issues (*e.g.* bugs, improvements, new features, tasks, and sub-tasks). Their commit messages typically contain corresponding issue number. For example, in Cassandra, a commit's message says "implement multiple index expressions. patch by jbellis; reviewed by Nate McCall for CASSANDRA-1157". In the issue tracker, the page of the issue[4] lists its description, the discussions among programmers, and the relations to other issues. In the Apache projects, when writing issue number to messages, programmers typically use the "name-number" pattern, where *name* denotes the name of a project, and *number* denotes the issue number. We rely on this pattern to extract issue number, and checks issue trackers to determine whether a commit is a bug fix.
2. Keyword. Issue trackers do not store all the bug fixes. In some cases, programmers may bypass issue trackers, especially when they believe that a change is trivial. When they commit a change, programmers may write a message to describe the fix. For example, in Aries, the message of a commit says "Fix broken service registration listener". We determine a commit as a bug fix, if its message contains words such as "bug" or "fix". The preceding commit was identified as a bug fix, since its message contains the keyword "fix". The heuristic is simple, and a number of previous studies (*e.g.* [15]) used the same technique to extract bug fixes.

Our underlying tool analyzes only source code (see Section 3.2 for details), so we ignore those bug fixes that do not modify code. In addition, we ignore bug fixes on test code, since their implementations are different from production code. Table 1 shows the collected bug fixes in our study. Column "Name" shows names of our subjects. All the projects are from the Apache software foundation[5], and are written in Java. Column "Bug Fix" lists number of collected bug fixes. Column "LOC" lists lines

[4] `https://issues.apache.org/jira/browse/CASSANDRA-1157`
[5] http://www.apache.org/

Table 1: Dataset

Name	Fix	LOC	File
aries	442	492,349	2,423
cassandra	2,463	9,683,554	19,918
derby	2,392	17,601,539	26,359
mahout	438	538,277	3,712
Total	5,735	28,315,719	52,412

of code. Column "File" lists number of files. In total, we collected more than five thousand bug fixes. The large quantity ensures the representativeness of our study.

3.2 GRAPA

It needs precise analysis to build such graphs, but due to many unknown code names, partial code analysis is typically inaccurate. In our previous work [50], we implement a tool, called GRAPA, that is able to build accurate delta graphs for partial code. The key insight of GRAPA is that we can leverage complete-code tools such as WALA to analyze partial code, if we fully fix unknown names for partial code. To achieve this goal, GRAPA extends the inference strategies of PPA [3]. In addition, many projects provide all their releases. For this example, the Derby project provides a page for downloading all its current and archive versions[6]. GRAPA implements a technique that compares used code elements with declared elements in releases to locate the context versions for a piece of partial code. With located versions, GRAPA fixes remaining unknown code names. GRAPA then integrates WALA to builds a dependency graph for each method.

Before building system dependency graphs, WALA translates source code into its internal representation (IR), which is similar to Java bytecode. As a result, in a system dependency graph of an `m` method, a node denotes an instruction in the IR. These nodes can be quite different from the source code. WALA generates a label for each node to describe its content. For example, a label can be:

```
invoke special...java/util/HashMap, <init>()V
```

Here, the node calls the default constructor of the `java.util.HashMap` class.

With the support of the Hungarian algorithm [16], GRAPA is able to detect the differences between two given graphs. Our previous work [52] show that GRAPA correctly detects the differences for more than 90% of fixes.

3.3 The Procedure

Given a fix f_b and a set of past fixes $F = \{f_1, \ldots, f_n\}$, we investigate whether f_b can be constructed from F by taking the following three steps:

Step 1. Building system dependency graphs for each pair of modified methods. For each fix, we first identify its pairs of modified methods, and build two system dependency graphs for each pair of modified methods $\langle g_l, g_r \rangle$. We use P to denote

[6] http://db.apache.org/derby/derby_downloads.html

the graph pairs that are built from a fix. Here, the definition of system dependency graphs is as follow:

Definition 1 A system dependency graph (SDG) is defined as $g = \langle V, E \rangle$, where V is a set of nodes, and $E \subseteq V \times V$ is a set of edges. Each node denotes an instruction in the IR of WALA. A $\langle v_1, v_2 \rangle$ edge denotes that there exists a data or control dependency from v_1 to v_2.

GRAPA enables WALA to build SDGs for bug fixes. Typically, a SDG is huge. When we analyze bug fixes, we focus on deltas, instead of whole graphs. As a result, we apply only intra-procedural analysis, when we build SDGs. The strategy reduces sizes of SDGs, but does not lose information. As we compare all methods, if called methods are modified, their deltas are extracted, when we analyze such called methods.

Step 2. Building delta graphs for different overlap metrics. Based on our previous study on bug fixes [51], we identify two types of changes:

1. Code structure changes involve modifications of program structures. For example, LUCENE-1510[7] says that the following code throws `NullPointerException`

```
  return new byte[0];
```

Programmers change its code structure, and add an `if` statement:

```
if (norms == null){
  return new byte[0];
}
```

2. Code name changes involve changes on code names. For example, CASSANDRA-2174 says that it fails to read saved files, and the faulty line is as follow:

```
in = new ObjectInputStream(...);
```

The above code call an incompatible API. To fix the bug, programmers choose another API, and the modified line was as follow:

```
in = new DataInputStream(...);
```

The above fix changes API code names. The other fixes can change client code names. For example, ARIES-1078 says that the buggy code checks a wrong value:

```
public Object addingService(ServiceReference ref){
   ServiceReference reference = ...
   if(ref)...
}
```

The fixed code checks the correct value:

```
public Object addingService(ServiceReference ref){
   ServiceReference reference = ...
   if(reference)...
}
```

Kim *et al.* [13] implement fix patterns both for code structure changes (*e.g.*, `obj.m()` $\rightarrow$ `if(obj!=null){obj.m()}` and for code name changes (*e.g.*, replacing a method call with any method calls, if they have compatible parameters).

After we build SDGs from source code, code structure changes refer to the graph structure, and code name changes refer to any change to a nodes label. To detect code changes, GRAPA builds delta graphs for two given SDGs, and the definition of a delta graph is as follow:

[7] https://issues.apache.org/jira/browse/LUCENE-1510. To save space, in the rest of the paper, we present only ids of fixes, but do not present their urls. All the fixes come from Apache, and their urls can be generated by replacing the id in the above url.

Definition 2 A delta graph is defined as a triple, $\delta = \langle SG_l, SG_r, L \rangle$, where SG_l is a set of subgraphs of a system dependency graph g_l; SG_r is a set of subgraphs of another system dependency graph g_r; and $L \subseteq G_l \times G_r$ is a set of edges. Each node of a delta graph is a node from a system dependency graph. A $\langle sg_{l1}, sg_{r1} \rangle$ edge denotes that sg_{l1} is modified to sg_{r1}.

In the above definition, each node in a graph is unique, and associated with a unique identification number. For two system dependency graphs, their delta graphs show their modified nodes and the mappings between such nodes. To compare g_l and g_r, we need to define the distance between their nodes (m in g_l and n in g_r). In the optimization research, the assignment problem [28] is to assign agents to their proper tasks, and the Hungarian algorithm is a classical algorithm that solves the assignment problem. Initially, the algorithm calculates the distances between agents and tasks. In each iteration, it selects a pair with the minimum distance, and updates the distances until all the pairs are selected. We consider the comparison of system dependency graphs as the assignment problem. For each pair of modified methods (m_l and m_r) in a bug fix, GRAPA builds two system dependency graphs, g_l and g_r. GRAPA compares g_l and g_r with the support of the Hungarian algorithm [16], and creates a delta graph. When GRAPA compares both code structures and code names, we define the distance as follow:

$$dis(m,n) = \frac{|i(m) - i(n)|}{i(m) + i(n)} + \frac{|o(m) - o(n)|}{o(m) + o(n)} + d(l(m), l(n)) \tag{1}$$

In this equation, $i(m)$ returns the indegree of m; $o(m)$ returns the outdegree of m; $l(m)$ returns m' label that is generated by WALA; and $d(l_1, l_2)$ returns the Levenshtein edit distance. Here, we calculate indegrees and outdegrees both on built system dependency graph. We notice that although two code snippets are identical, their graphs can be different, since their locations can be different. To detect such similar code, we remove identifications of statements and locations from generated labels before comparing graphs. We consider that two nodes are matched, if their distance is zero by Equation 1. We merge all the delta graphs of each fix into a delta graph δ.

When we compare only code structures, we revise the distance function to compare delta graphs:

$$dis(m,n) = \frac{|i(m) - i(n)|}{i(m) + i(n)} + \frac{|o(m) - o(n)|}{o(m) + o(n)} + d(\mu(m), \mu(n)) \tag{2}$$

If we compare only edges, it can produce false mappings, since two different types of nodes can have the same input and output edges. To reduce the false mappings, in the above equation, we compare types of nodes with $d(\mu(m), \mu(n))$. Here, $\mu(m)$ is defined in Equation as follow:

$$\mu(v) = \begin{cases} \text{invoke method}, & v \text{ invokes a method.} \\ \text{get field}, & v \text{ gets a field.} \\ \text{put field}, & v \text{ puts a field.} \\ type(v), & \text{otherwise.} \end{cases} \tag{3}$$

Table 2: Overlap metrics

Reusable Component	Strategy
Both structure and code changes	FI. Searching for the best single match. PI. Searching for the maximum coverage.
Code structure changes	FS. Searching for the best single match. PS. Searching for the maximum coverage.
Code name changes	FN. Searching for the best single match. PN. Searching for the maximum coverage.

Here, $type(v)$ returns the type of a node that is defined by WALA[8].

Although we do not intend to define an abstract graph, the above equation transfers delta graphs into a more abstract format. A specific repair tool can have a different abstraction granularity. Indeed, to the best of our knowledge, existing tools cannot synthesize patches only based on our abstract granularity. However, our study does not evaluate the repair capabilities of specific tools. Instead, based on our abstraction, our study provides evidences to answer the open questions as listed in Section 2, and our corresponding results provide an upper bound for the future repair approaches, if they reuse past fixes to synthesize patches.

When we compare only code names, the distance function is revised as follow:

$$dis(m,n) = d(l(m), l(n)) \tag{4}$$

For the two metrics, we merge all the delta graphs of each fix into a delta graph β.

Step 3. Comparing delta graphs for overlaps. We revise Equation 1 to compare two delta graphs:

$$dis'(m,n) = \begin{cases} \text{dis(m,n)}, & \text{m, n are on the same side.} \\ \infty, & \text{otherwise.} \end{cases} \tag{5}$$

For different metrics, $dis(m,n)$ is defined in Step 2. Each delta graph consists of two sides (SG_l and SG_r). The modification ensures that nodes of the two sides cannot be wrongly matched.

3.4 Overlap Metrics

As shown in Table 2, when defining our overlap metrics, we envision three types of reusable components: (1) code structure changes, (2) code names changes, or (3) their combinations, and we imagine two ways to leverage the reusable components: (1) finding a matching component, or (2) merging several components. We next formally define our overlap metrics.

Suppose that there are a set of past fixes $F = \{f_1, \ldots, f_n)\}$, and a new bug fix f_b. Their delta graphs are $\Delta = \{\delta_1, \ldots, \delta_n\}$, and δ_b. Intuitively, if δ_b is a subgraph of $\delta_i (i \in [1..n])$, or a super graph which can be composed of subgraphs in Δ, f_b can be generated from F. In our empirical study, we use GRAPA to compare bug fixes, and define six overlap metrics in correspondence with the six strategies. Ideally, if f_b is

[8] `http://wala.sourceforge.net/javadocs/trunk/`

similar to F according to a specific metric, the corresponding strategy will effectively generate f_b from F. We next present the formal definitions of our overlap metrics:

1. Fully overlapped bug fixes (FI): A previous fix δ_i covers both the structure and name changes of δ_b, *i.e.*, $\delta_b \overset{I}{\subseteq} \delta_i$. For example, we find that $\delta_{CASSANDRA-5644} \overset{I}{\subseteq} \delta_{CASSANDRA-6258}$, and Figure 1a shows the overlapped fixes. As for the above two fixes, CASSANDRA-5644 complains a swallowed exception, and CASSANDRA-6258 complains that the root cause of an exception is not presented. Our result shows that the two related bugs have common fixes. The two fixes in Figure 1a are fully identical (FI), since their delta graphs are the same.

2. Partially overlapped bug fixes (PI): No previous fix can cover both the structure and name changes of δ_b, but the composition of some fixes cover both types of changes, *i.e.*, $\delta_b \overset{I}{\subseteq} \delta_m \cup \ldots \cup \delta_n$. For example, we find that $\delta_{CASSANDRA-4432} \overset{I}{\subseteq} \delta_{CASSANDRA-6822} \cup \delta_{CASSANDRA-4925}$. As shown in Figure 2b. In CASSANDRA-4432, programmers modified two method invocations. The first change is identical with the changes in CASSANDRA-6822, and the second change is identical with the changes in CASSANDRA-4925.

With Equation 3, we transfer delta graphs into abstract graphs as $A = \{\alpha_1, \ldots, \alpha_n\}$, and α_b. We define the following two overlap metrics relevant to structure changes:

3. Fully overlapped structure changes (FS): The structure changes of a previous fix α_i cover the structure changes of α_b, *i.e.*, $\alpha_b \overset{S}{\subseteq} \alpha_i$. For example, $\alpha_{CASSANDRA-4279} \overset{S}{\subseteq} \alpha_{CASSANDRA-6618}$, as shown in Figure 2a. Although the their delta graphs are different, after we use Equation 3 to map the labels, all the delta graphs are reduced to the same abstract graph.

4. Partially overlapped structure changes (PS): α_b is composed of known structure changes, *i.e.*, $\alpha_b \overset{S}{\subseteq} \alpha_m \cup \ldots \cup \alpha_n$. For example, $\alpha_{ARIES-1304} \overset{S}{\subseteq} \alpha_{ARIES-682} \cup \alpha_{ARIES-703}$, as shown in Figure 2b. In ARIES-1304, the first fix adds a method invocation, and the second fix modifies a method invocation. The structure of the first fix can be constructed from ARIES-682, and the structure of the second fix can be constructed from ARIES-703.

Bug fixes can involve code name changes. We define a function $\theta(\delta)$ to collect code name changes:

$$\theta(\delta) = \{(name_o, name_n)\} \tag{6}$$

where $name_o$ denotes an original code name, and $name_n$ denotes its modified new code name. For F and f_b, the extracted names changes are represented as $B = \{\beta_1, \ldots, \beta_n\}$ and β_b. The following two overlap metrics are defined for name changes:

5. Fully overlapped name changes (FN): The name changes of a previous fix β_i cover the name changes of β_b, *i.e.*, $\beta_b \overset{N}{\subseteq} \beta_i$. For example, although in Figure 3a, CASSANDRA-1681 and CASSANDRA-1701 do not the same code structure changes, both replace an API, `array()` with the correct API, `byteBufferToByteArray()`.

6. Partially overlapped name changes (PN). β_b is composable of known name changes, *i.e.*, $\beta_b \overset{N}{\subseteq} \beta_1 \cup \ldots \cup \beta_n$. For example, in Figure 3b, the code name changes of

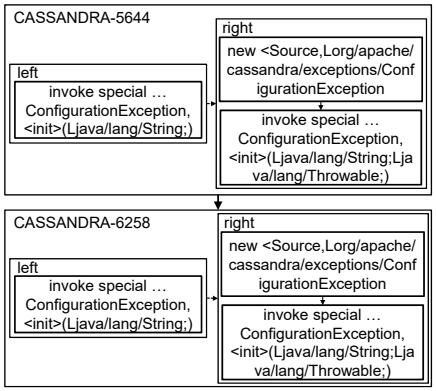
CASSANDRA-5644 (δ_b)

```
catch (SocketException e)
{
-throw new ConfigurationException("...");
+throw new ConfigurationException
+("...",e);
}
```

CASSANDRA-6258 (δ_i)

```
catch (ClassNotFoundException e)
{
-throw new ConfigurationException("...");
+throw new ConfigurationException
+("...",e);
}
```

The delta graphs



(a) FI ($\delta_b \overset{I}{\subseteq} \delta_i$)

CASSANDRA-4432 (α_b)

```
- ksm.toSchema(
-    System.currentTimeMillis());
+ ksm.toSchema(
+   FBUtilities.timestampMicros());
...
- announce(..., System.nanoTime());
+ announce(...,
+   System.currentTimeMillis());
```

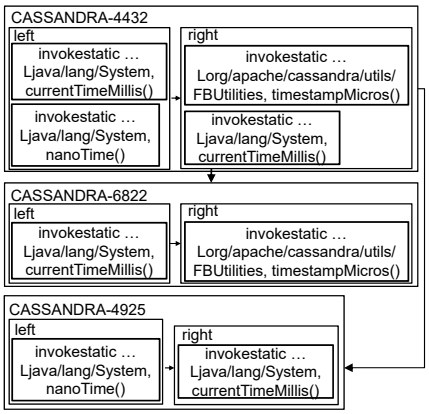
CASSANDRA-6822 (α_2)

```
- mutation.delete(...,
-    System.currentTimeMillis());
+ mutation.delete(...,
+    FBUtilities.timestampMicros());
```

CASSANDRA-4925 (α_1)

```
- long now=System.nanoTime();
+ long now=System.currentTimeMillis();
```

The delta graphs



(b) PI ($\delta_b \overset{I}{\subseteq} \delta_1 \cup \delta_2$)

Fig. 1: Sample code for FI and PI

CASSANDRA-2625 can be constructed from CASSANDRA-536, CASSANDRA-1829, and CASSANDRA-2057, *i.e.*, $\beta_{CASSANDRA-2625} \overset{N}{\subseteq} \beta_{CASSANDRA-536} \cup \beta_{CASSANDRA-1829} \cup \beta_{CASSANDRA-2625}$.

In summary, FI and PI concern both code structure changes and name mappings. FS and PS concern only code structure changes; and FN and PN concern only code name mappings. FI, FS, and FN compare individual fixes, and PF, PS, and PN compare multiple fixes.

Our overlap metrics are not exclusive nor symmetric. Indeed, our study does not aim to classify bug fixes into exclusive categories, but explores the potential of using hints from past fixes. When defining our overlap metrics, we consider two aspects of

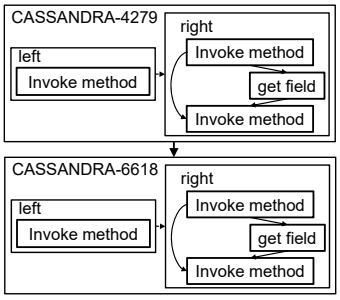
CASSANDRA-4279 (α_b)

```
this.minCompactionThreshold.set(...);
+ CompactionManager.instance.
+     submitBackground(this);
```

CASSANDRA-6618 (α_i)

```
connectionTracker.allChannels.add(...);
+ isRunning.set(true);
```

The structure changes



(a) FS ($\alpha_b \overset{S}{\subseteq} \alpha_i$)

ARIES-1304 (α_b)

```
-for(... mbeanServers){
+for(... mbeanServers.keySet()){
...
-mbeanServers.add(server);
+mbeanServers.put(server,Boolean.TRUE);
```

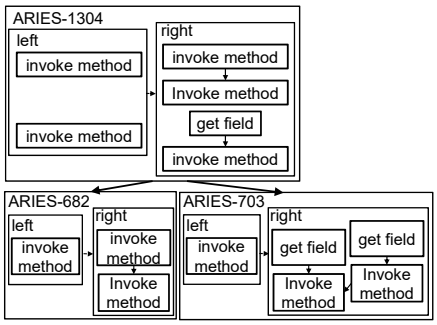
ARIES-682 (α_1)

```
-else name=file.getAbsolutePath()
-   .substring(...);
+else name=file.getAbsolutePath()
+   .substring(...).replace(...);
```

ARIES-703 (α_2)

```
-value=recipe.create();
+value=convert(recipe.create(),
+    conversionType);
```

The structure changes



(b) PS ($\alpha_b \overset{S}{\subseteq} \alpha_1 \cup \alpha_2$)

Fig. 2: Sample code for FS and PS

repair capabilities in future repair approaches. First, a repair approach can use either single fix or multiple fixes to construct a new fix. We define fully overlapped and partial overlapped (F and P) for this capability. Second, a repair approach can use repetitive code changes, only structure changes, or only code name changes, when it constructs new fixes. We define I, S, and N to denote the three types of repair capabilities. The relationship of I, S, and N is as follow: $\delta_b \overset{I}{\subseteq} \delta_m \cup \ldots \cup \delta_n \Rightarrow \delta_b \overset{S}{\subseteq} \delta_m \cup \ldots \cup \delta_n \wedge \delta_b \overset{N}{\subseteq} \delta_m \cup \ldots \cup \delta_n$, but $\delta_b \overset{S}{\subseteq} \delta_m \cup \ldots \cup \delta_n \wedge \delta_b \overset{N}{\subseteq} \delta_m \cup \ldots \cup \delta_n \not\Rightarrow \delta_b \overset{I}{\subseteq} \delta_m \cup \ldots \cup \delta_n$, since the mappings of nodes in S and N can be different. For bug fixes without code name changes, $\delta_b \overset{I}{\subseteq} \delta_m \cup \ldots \cup \delta_n \Leftrightarrow \delta_b \overset{S}{\subseteq} \delta_m \cup \ldots \cup \delta_n$, since $\delta_b \overset{N}{\subseteq} \delta_m \cup \ldots \cup \delta_n$ always holds. Similarly, for bug fixes without code structure changes, $\delta_b \overset{I}{\subseteq} \delta_m \cup \ldots \cup \delta_n \Leftrightarrow \delta_b \overset{N}{\subseteq} \delta_m \cup \ldots \cup \delta_n$, since $\delta_b \overset{S}{\subseteq} \delta_m \cup \ldots \cup \delta_n$ always holds. Meanwhile, S and N is irrelevant, since they focus on two different types of

CASSANDRA-1681 (β_b)

```
-return new BigInteger(bytes.array());
+return new BigInteger(TBaseHelper.
+     byteBufferToByteArray(bytes));
```

CASSANDRA-1701 (β_i)

```
-getValidatorForValue(
-  ..., columnNameInBytes.array());
+getValidatorForValue(...,TBaseHelper.
+     byteBufferToByteArray(columnName));
```

The code name changes

CASSANDRA-1681:
1. $\emptyset \rightarrow$ `TBaseHelper.byteBufferToByteArray`
2. `array` $\rightarrow \emptyset$

CASSANDRA-1701:
1. $\emptyset \rightarrow$ `TBaseHelper.byteBufferToByteArray`
2. `array` $\rightarrow \emptyset$
3. `columnNameInBytes` $\rightarrow$ `columnName`

(a) FN ($\beta_b \overset{N}{\subseteq} \beta_i$)

CASSANDRA-2625 (β_b)

```
+ SystemTable.setBootstrapped(true);
+ setToken(token);
```

CASSANDRA-536 (β_1)

```
+ SystemTable.updateToken(token);
```

CASSANDRA-1829 (β_2)

```
+ SystemTable.setBootstrapped(true);
```

CASSANDRA-2057 (β_3)

```
+ setToken(getLocalToken());
```

The code name changes

CASSANDRA-2625:
1. $\emptyset \rightarrow$ `SystemTable.setBootstrapped`
2. $\emptyset \rightarrow$ `setToken`
3. $\emptyset \rightarrow$ `token`
4. $\emptyset \rightarrow$ `true`

CASSANDRA-536:
1. $\emptyset \rightarrow$ `SystemTable.updateToken`
2. $\emptyset \rightarrow$ `token`

CASSANDRA-1829:
1. $\emptyset \rightarrow$ `SystemTable.setBootstrapped`
2. $\emptyset \rightarrow$ `true`

CASSANDRA-2057:
1. $\emptyset \rightarrow$ `setToken`
2. $\emptyset \rightarrow$ `getLocalToken`

(b) PN ($\beta_b \overset{N}{\subseteq} \beta_1 \cup \beta_2$)

Fig. 3: Sample code for FN and PN

code changes, *i.e.*, $\delta_b \overset{S}{\subseteq} \delta_m \cup \ldots \cup \delta_n \nLeftrightarrow \delta_b \overset{N}{\subseteq} \delta_m \cup \ldots \cup \delta_n$. In total, the combination of the above two aspects produces six overlap metrics (*e.g.*, FI and PI).

Our overlap metrics do not consider the contexts and semantics of deltas, although we agree that such information is useful to locate useful fixes. However, no matter how an approach repairs bugs, its synthesized fixes shall follow our overlap metrics, if it reuses known fixes. When comparing structure changes, we have to define the transfer function (Equation 3). We understand that the granularity is not aligned to a specific approach. To the best of our knowledge, our granularity is coarser than any existing approaches. Here, we positively believe that some future approaches can achieve our granularity, and we provide an estimation for their potentials. We next present our support tool that detects overlapped bug fixes for different metrics.

We envisage that future automatic-program-repair approaches can fully or partially implement the components in Table 2. Before researchers realize such approaches, our empirical results are useful for them to estimate the potential of their proposed approaches. For example, if a proposed approach uses only one past fix at a time and it needs changes of both code structures and code names from past fixes, we can use the results of FI to estimate the repair potential of such an approach. As another

Table 3: Overall result of learning from the same project (Part 1)

Project	Both				Fix
	FI	%	PI	%	
aries	8	1.8%	10	2.3%	442
cassandra	68	2.8%	115	4.7%	2,463
derby	37	1.5%	44	1.8%	2,392
mahout	9	2.1%	14	3.2%	438
Total	122	2.1%	183	3.2%	5,735

Table 4: Overall result of learning from the same project (Part 2)

Project	Structure				Code Name			
	FS	%	PS	%	FN	%	PN	%
aries	38	8.6%	144	32.6%	16	3.6%	37	8.4%
cassandra	383	15.6%	1,202	48.8%	126	5.1%	327	13.3%
derby	249	10.4%	865	36.2%	63	2.6%	169	7.1%
mahout	47	10.7%	155	35.4%	12	2.7%	29	6.6%
Total	717	12.5%	2,366	41.3%	217	3.8%	562	9.8%

example, if a proposed approach can fully resolve code structure changes from other sources and needs only code name changes from past fixes, we can estimate its potential, based on the results of FN and PN.

Our defined overlap metrics do not cover all the relationships between new fixes and past fixes, but it is feasible to define more overlap metrics that cover more types of relationships. For example, a future approach can focus on constructing a specific type of bug fixes (*e.g.*, API-related bug fixes) from past fixes. Before such an approach is proposed, we can estimate its potential by defining another overlap metric, and such a metric corresponds to another relationship between new fixes and past fixes.

4 Empirical Result

In this section, we present our answers to the four open questions in Section 2. For a project, in each iteration, we consider one of its fixes as a fix under analysis. For each fix under analysis, in Sections 4.1, 4.2, and 4.3, we consider all the *other* fixes of the same project as past known fixes; and in Section 4.4, we consider all the fixes of the other projects as past known fixes. As introduced in Section 3.3, for each fix under analysis, we compare its delta graphs with the delta graphs of all the past known fixes. The follow-up sections present our results.

4.1 OP1. Repair Potential

Tables 3 and 4 show the overall result[9]. Column "Project" lists names of projects. Columns "Both", "Structure", and "Code Name" list matched bugs with corresponding overlap metrics. Column "Fix" lists number of collected fixes. In total, Column "Both" shows that only several percents of bug fixes can be constructed from past

[9] PI is more restricted than PS and PN, but not the combination of the two. As a result, Column PI is not the sum of Columns PS and PN.

fixes, if an approach requires both structure changes and code name mappings from past fixes. Column "Code Name" shows slightly better results, but Column "Structure" shows that more bug fixes can be constructed, if a repair approach requires only structure changes from past fixes. Here, as discussed in the end of Section 3.4, we assume that such an approach can fully resolve the missing information from other sources. The results lead to our first finding:

Finding 1. In total, there are 2.1% FI, 12.5% FS, and 3.8% FN similar bug fixes, indicating that if an repair approach only reuses the contents, the structure changes, or name changes from past fixes, at most 2.1%, 12.5%, and 3.8% new bugs can be constructed in such ways, correspondingly.

If an approach needs both structure changes and code name mappings from past fixes, our results show that only 2.1% bugs can be fixed in this way.

Column "Structure" shows that PS corresponds to more fixed bugs than FS, and Columns "Both" and "Code Name" show that PI and PN also have more fixed bugs than FI and FN, respectively, but the difference is less significant. The results lead to our second finding:

Finding 2. In total, there are 3.2% PI, 41.3% PS, and 9.8% PN similar bug fixes. Compared with Finding 1, the improvements implies that a repair approach that combines multiple fixes is promising to repair much more bugs, especially for learning code structure changes.

In summary, only about 2.1% of bugs can be fixed, if a repair approach requires identical matches from past fixes. Composing multiple fixes does not improve the results. However, if a repair approach requires only structure changes from past fixes, it can fix about 10% bugs, and composing multiple can fix 30% of bugs.

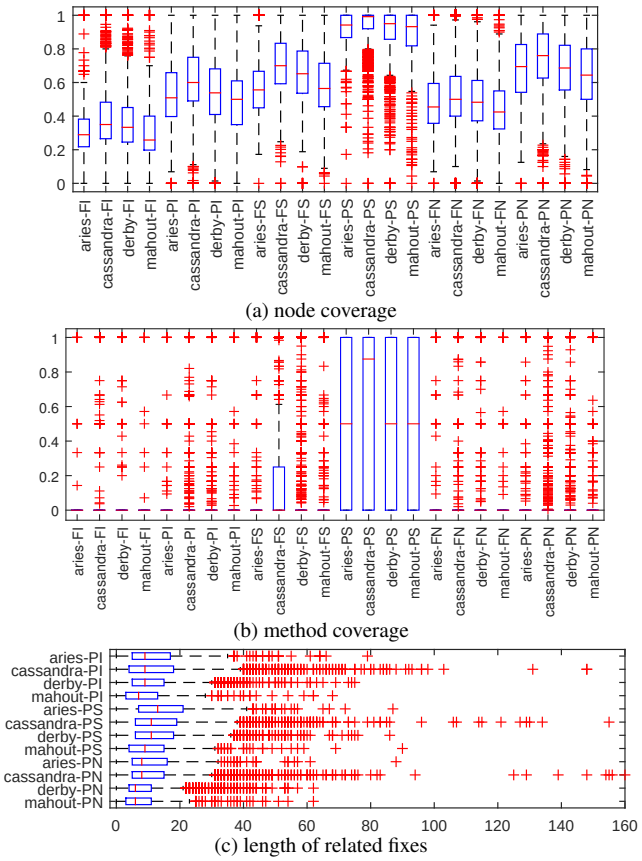
4.2 OP2. Creativity in Fixing Bug

Section 4.1 shows that many bugs cannot be fully fixed, and they have unmapped nodes. If a method has unmapped nodes, we consider the method as unfixed. We measure the challenges in fixing a bug with the metrics such as node coverage, method coverage, and candidate length as follows:

1. Node coverage presents the coverage of matched nodes:

$$node\ coverage(f) = \frac{|mapped\ nodes|}{|total\ nodes|} \tag{7}$$

Figure 4a shows boxplots of our results. The horizontal axis lists the combinations of projects and overlap metrics. For example, "aries-FI" denotes the node coverage, when we use FI to compare fixes of aries. The vertical axis lists corresponding node coverage. Figure 4a shows that about thirty percents of nodes can be covered, if we require both structure changes and code mappings, and their is an improvement of about ten percents, if we combine multiple fixes; about eighty percents of the nodes can be covered, if we consider only structure changes, and most nodes can be covered, if we combine multiple fixes; and about half of nodes can be covered on average, if we



(a) node coverage

(b) method coverage

(c) length of related fixes

Fig. 4: The results for OP2

consider only code mappings, and there is an improvement of about twenty percents, if we consider multiple bug fixes. The results lead to our third finding:

Finding 3. Many structure changes can be learnt from past fixes (50% for individual fixes, and 90% when for multiple fixes), while fewer code mappings can be learnt from past fixes (40% for individual fixes, and 70% for multiple fixes).

Kim *et al.* [13] manually extract fix patterns from thousands of bug fixes. Our results are largely consistent with their results, since most their extracted fix patterns are related to structure changes. Although their fix patterns do not include code name mappings, these patterns provide insights on fixing bugs, when such mappings are unavailable. For example, their method-replacer pattern replaces a method call with another method whose parameters and return type are compatible. This pattern simply tries all the compatible methods, and tries whether fixed code can pass test cases. It is feasible to fix specific bugs, but does not cover all code name mappings. For example, the code mapping in Section 4.1 has incompatible parameters. As a result,

the fix pattern cannot replace the method call correctly, and their approach cannot fix the two bugs such as CASSANDRA-1681 and CASSANDRA-1701.

Finding 3 is consistent with prior studies (*e.g.*, [31]), since they find many repetitive changes. However, Finding 3 reveals that most bug fixes consist of both repetitive changes and non-repetitive changes, since Figure 4 shows that only outliers are fully repetitive. As a result, Findings 1 and 2 show that only a small portion of bug fixes can be fully constructed from past fixes.

2. Method coverage is defined as follow:

$$method\ coverage(f) = \frac{|overlapped\ methods|}{|buggy\ methods|} \tag{8}$$

Figure 4b shows the boxplots of our results. Its horizontal axis lists the combinations of projects and overlap metrics, and its vertical axis lists corresponding method coverage. Compared with Figure 4a, Figure 4b shows that about 30% of edited nodes are mapped between fixes, if we require both mappings of code structures and identifier names between every two fixes. However, if we allow one fix to partially match multiple fixes in terms of both code structures and identifier names, 10% more edited nodes are matched. By requiring only code structure mappings between every two fixes, we get 80% edited nodes mapped. If we allow one fix to partially match multiple fixes in terms of code structures, we can increase the percentage further by 20%. The result leads to our fourth finding:

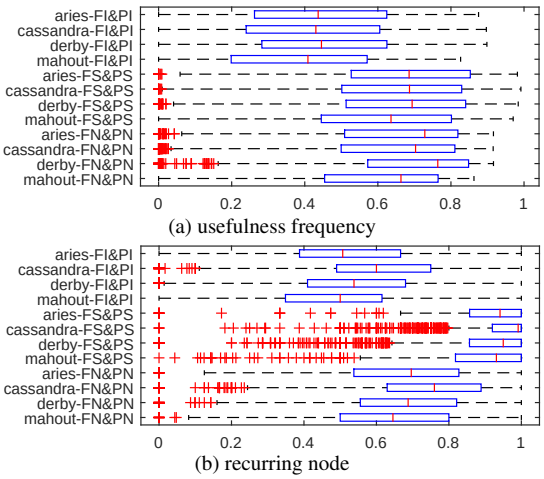
Finding 4. As matched nodes and unmatched nodes are evenly distributed in methods, a fix typically has both recurring fixes and creative fixes.

Nguyen *et al.* [32] show that many bug fixes are recurring. Figure 4a is largely consistent with their results, since most fixes have matched nodes with other fixes. However, our results reveal that at the method level, only a small portion of bug fixes are recurring.

3. Length of related fixes presents the length of fixes that have matched nodes with the fix under analysis. As introduced in Section 3.4, PI, PS, and PN learn from multiple related fixes. If a bug needs more useful fixes, it can be more difficult to fix the bug, since it needs to extract useful nodes from more related fixes. Here, we add a known fix into the related-fix category, only when the fix has additional matched nodes than previous known fixes. Figure 4c shows the boxplots of our results. We find that the medians of all the overlap metrics are around ten, but all the overlap metrics have exceptional points, where a bug has tens and even more than a hundred fixes. The result leads to our fifth finding:

Finding 5. In most cases, a bug has only about 10 useful fixes, but some exceptional bugs have tens of useful fixes.

In summary, it is feasible to learn most code structure changes, but it is unlikely to learn many code name mappings from past fixes. At the method level, a bug typically has both recurring fixes and creative fixes that never appear in previous fixes. Furthermore, a bug typically has quite limited fixes that can contribute to its fix.



(a) usefulness frequency

(b) recurring node

Fig. 5: The results for OP3

4.3 OP3. Preparing Fix Repository

In this section, we analyze the challenges in learning past fixes. We measure the challenges in fixing a bug with the metrics such as usefulness frequency, and recurring nodes as follows:

1. Frequency of useful past fixes present the value of a past fix:

$$usefulness\ frequency(f) = \frac{|related\ fixes|}{|total\ known\ fixes|} \tag{9}$$

This metric analyzes the challenges from the side of past known fixes. For a known fix f, we calculate the metric as its related fixes over total past fixes. As introduced before, we consider that two fixes are related, if they have matched nodes. Figure 5a shows the boxplots of our results. Its vertical axis lists the combinations of projects and overlap metrics, and its horizontal axis lists corresponding usefulness frequency. The result shows that for structure changes, the usefulness frequency are about ninety percents; for code name mappings, the usefulness frequency are about seventy percents; and for both structure changes and code name mappings, the usefulness frequency are about forty percents. The result leads to our sixth finding:

Finding 6. If we focus on only structure changes or code name mappings, past fixes have high usefulness frequency, no matter if we learn from the same project (70%), or other projects (40%).

2. Recurring nodes present the recurring nodes of a past fix:

$$recurring\ node(f) = \frac{|recurring\ nodes|}{|total\ nodes|} \tag{10}$$

For a known fix, recurring nodes are its nodes that are matched with other fixes, and total nodes are its total number of nodes. Figure 5b shows the boxplots of our

Table 5: Overall result of learning from other project (Part 1)

Project	Both				Fix
	FI	%	PI	%	
aries	1	0.2%	3	0.7%	442
cassandra	30	1.2%	44	1.8%	2,463
derby	13	0.5%	15	0.6%	2,392
mahout	6	1.4%	9	2.1%	438
Total	50	0.9%	71	1.2%	5,735

Table 6: Overall result of learning from other project (Part 2)

Project	Structure				Code Name			
	FS	%	CS	%	FN	%	PN	%
aries	49	11.1%	109	24.7%	4	0.9%	7	1.6%
cassandra	353	14.3%	813	33.0%	39	1.6%	52	2.1%
derby	252	10.5%	646	27.0%	18	0.8%	25	1.0%
mahout	57	13.0%	133	30.4%	10	2.3%	11	2.5%
Total	711	12.4%	1,701	29.7%	71	1.2%	95	1.7%

results. Its horizontal axis lists the combinations of projects and overlap metrics, and its vertical axis lists corresponding recurring nodes. The result in Figure 5b leads to the seventh finding:

Finding 7. When we learn from bug fixes of the same project, 90% or 70% of nodes are recurring if we consider only structure changes or only code name mappings, respectively. Even if we consider both structure changes and code name mappings, 60% of nodes are recurring.

In summary, when learning from the same projects, most fixes are somewhat useful. Some common structure changes and code name mappings appear in more than one fix.

4.4 OP4. Learning from Other Project

1. Overall results. Tables 5 and 6 show the overall results of learning from other projects. The columns in Tables 5 and 6 are of the same meanings with Tables 3 and 4. Table 1 shows that aries and mahout both have much fewer fixes than cassandra and derby. As a result, when learning from other projects, for a fix under analysis, aries and mahout have much more past known fixes to compare, and the past known fixes of cassandra and derby do not change as much. The difference allows us investigating the impacts of more past fixes.

Column "Structure" shows that with much more past fixes, for FS, the matched fixes of aries and mahout increase several percents. Although their past known fixes also increase, the matched fixes of cassandra and derby both decrease. For PS, the matched fixes of all the projects decrease. The results indicate that it is more difficult to learn multiple fixes. Column "Code Name" shows that for FN and PN, the matched fixes of all the projects decrease. The results show that it is much more difficult to learn code mapping from other projects. As a result, Column "Both" shows that when learning from other projects, although past known fixes become much more, fewer bugs can be matched. The result leads to our eighth finding:

Finding 8. Overall, it is more challenging to learn from other projects than from the same projects.

2. Challenges in fixing bugs. To present details, we calculate the node coverage and method coverage when learning from other projects. Figures 6a and 6b show the boxplots of the two metrics, respectively. As shown in Equation 3, FN transfers code names to abstract names (*e.g.*, field and method) that are independent on projects. The two figures show that the medians of FS and FN do not change much. The result indicates that most structure changes can be learnt from other projects. The two figures show that the medians of FN and PN decrease. Based on the availability of source code, we identify two types of code names such as client-code names and API code names. For client code, projects often have different names, so it is infeasible to mine such mappings from other projects. For API code names, projects can use different API libraries, so it can be difficult to mine such mappings either. In total, only about two percents of fixes have code name mappings that appear in other projects. After manual inspection, we find most such mappings define frequent API elements in popular API libraries (*e.g.*, J2SE). For example, the three bugs in Section 4.1 all require code mappings of J2SE. As J2SE is widely used, we find that bug fixes in other projects also require its code name mappings. For example, CASSANDRA-3751 says that dead locks occur when committing logs, and the buggy code is as follow:

```
Set<Table> tablesRecovered = new HashSet<Table>();
```

The fixed code replaces the called API method:

```
Set<Table> tablesRecovered = new NonBlockingHashSet<Table>();
```

We suspect that the mappings of client code are difficult to be learnt. For example, in Section 4.1, the desirable code name mapping of CASSANDRA-1701 involves a method that is declared in `TBaseHelper` class. The class is implemented by the programmers of cassandra. As other projects do not implement the same class, it is infeasible to learn such a mapping from their fixes. The results lead to our ninth finding:

Finding 9. When we learn fixes from other projects, compared with from the same project, most structure changes can still be learnt; some API code mappings can be learnt; but client code mappings can hardly be learnt.

We further calculate lengths of related fixes, and Figure 6c shows the boxplots of our results. We find that their results in Figure 4c and Figure 6c are quite similar. Although aries and mahout have much more past known fixes when learning from other projects, Figure 6c shows that the medians of the two projects do not increase consequently. Figure 4c and Figure 6c show that for all the projects, most fixes have about ten useful past fixes. The results lead to our tenth finding:

Finding 10. For most fixes, from certain point, their useful past fixes do not increase with more past fixes.

This finding says that most bug fixes are limited useful, and more past fixes may not help. This limitation can touch our second open question, *i.e.*, the creativity of fixing bugs.

3. Challenges in learning from past fixes. To present challenges in learning from past fixes of other projects, we calculate usefulness frequency and recurring node.

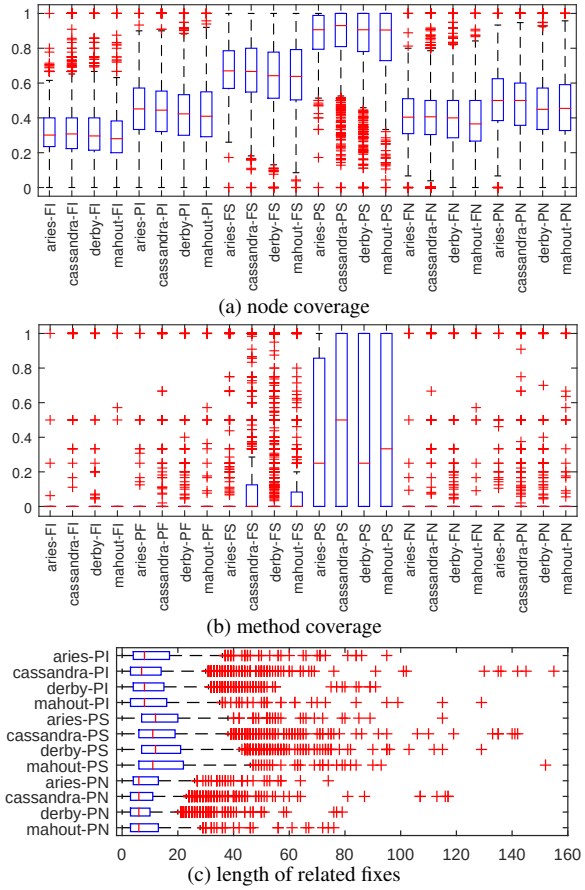


Fig. 6: OP4. Replicating OP2 with other projects.

Figures 7a shows the boxplots of usefulness frequency. We find that for this metric, cassandra and derby do not change much, but aries and mahout become much smaller. Table 1 shows that aries and mahout have much fewer fixes than cassandra and derby, and cassandra has a close number of fixes with derby. Consequently, the known fixes of aries and mahout become much larger than those in Figures 5a, and the known fixes of cassandra and derby do not change much. Here, the upper binds of the usefulness frequency metric are below one, when learning from other projects. This result leads to our finding:

Finding 11. The usefulness frequency decreases significantly with the increasing of past known fixes.

Figure 7b shows the boxplots of recurring node. We find that Figure 7b and Figure 5b are quite similar. Figure 7a shows that the usefulness frequency of aries and mahout become much smaller, but Figure 7b shows that the recurring nodes of the two projects do not change much. The results lead to our twelfth finding:

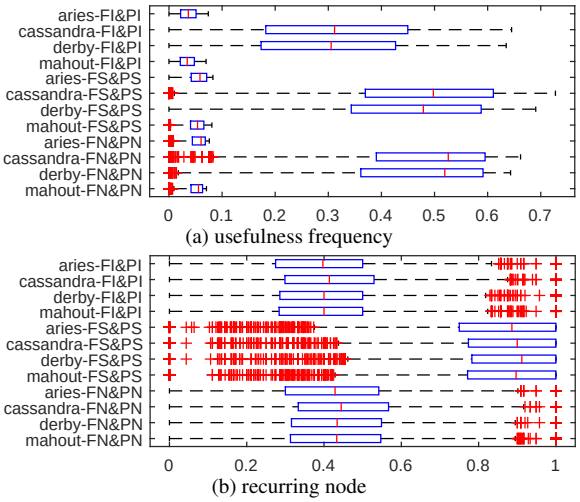


Fig. 7: OP4. Replicating OP3 with other projects.

Finding 12. Most fixes have matched nodes with other fixes, and a few common structure changes and code name mappings may exist in many fixes.

Nguyen *et al.* [32] show that recurring fixes exist in the same projects. Figure 7b indicates that even across projects, some fix actions are recurring, but comparing Figure 7b with Figure 5b, we find that across projects, recurring fix actions are fewer. The result is consistent with the empirical study of Barr *et al.* [1], which show that there are significantly more recurring fixes within projects than across projects.

In summary, it is more challenging to learn from other projects than from the same projects. Almost the same percents of structure changes can be learnt from past fixes, but it learns code name mapping for only about one percent of bugs. By composing multiple fixes, it is feasible to learn more structure changes, but it shows little improvement on learning code name mappings. When learning from other projects, past fixes increase significantly, but the useful fixes for a bug remain almost unchanged. As a result, it becomes much more difficult to locate useful fixes of other projects. A few common structure changes and code name mappings may appear in many fixes, but at the method level, it needs creative fixes, since recurring changes do not cover whole fixes.

4.5 Threats to Validity

The threat to internal validity includes the possible errors in extracting delta graphs. To reduce the threat, we build our tool on WALA, a mature analysis tool for Java, but even WALA can produce errors. The threat could be reduced if we leverage more advanced tools in future work. The threat to internal validity also includes that we do not consider the order of bug fixes, so we can overestimate the usefulness of past fixes. We notice that previous studies (*e.g.*, [1]) also share the same threat. Furthermore, in literature, the n-fold cross validation [7] is a common technique to

evaluate the effectiveness of a trained model. During the validation, researchers randomly partition their dataset into n equal sized subsamples. In each iteration, a single subsample is retained as the validation data for testing the trained model, and the remaining n-1 subsamples are used as the training data. Although the training data and validation data are not in a chronological sequence, it is widely used in various research fields including software engineering. In recent years, researchers start to explore the impacts of the time order empirically. For example, Tan *et al.* [39] conducted an empirical study to explore the effectiveness of defect prediction approaches. Their results show that in an extreme case, the fscore is significantly reduced from 0.54 to 0.073. It is worthy fully exploring the impact of the time order in our future, but we have to overcome several challenges. For example, for the in-project bug fixing suggestion scenario, it is difficult to uniformly define a timeline before which all fixes are leveraged to infer the fixes applied later. In addition, for the cross-project bug fixing suggestion scenario, since we can always extract a large number of diverse fixes from open source projects, our current usage of fixes ignoring time ordering can properly emulate the usage of past fixes from different projects. The overestimate problem can be mitigated in the cross-project bug fixing suggestion scenario, when various fixes can be possibly extracted from other projects. The threat to external validity includes that although we analyze thousands of fixes, our approach is evaluated on limited projects. The threat could be further reduced by analyzing more fixes in future work.

5 Discussion and Future Work

Constructing from multiple fixes. Our study reveals that identical fixes are rare, and it is more promising to learn from multiple fixes. Although Long and Rinard [23] mainly locate and reuse a single fix to repair a new bug, it can be feasible to extend their approach for reusing multiple fixes. In particular, their extended probabilistic model has to rank multiple useful fixes, and their extended feature extraction has to learn from multiple fixes. Besides the direct usage, it is feasible to mine and reuse patterns from multiple fixes. For example, when Kim *et al.* [13] use more than one pattern to repair a bug, their fix is in fact constructed from multiple fixes. Despite the potential, existing approaches are still insufficient to compose many fixes, even if they are overlapped with past fixes. Kaleeswaran *et al.* [11] extract repair hints from past fixes, and leave actual repairs to programmers. In future work, we plan to utilize their hints and our insights, and work on approaches that better reuse multiple fixes.

Constructing creative fixes. Tables 3 and 5 show that most fixes cannot be constructed solely from past fixes. Researchers applied the generation-and-validation strategy to construct creative repair actions. For example, Kim *et al.* [13] replace a method call with arbitrary method calls, if they have compatible parameters. The strategy can generate many candidates, so existing approaches suffer from the huge search space. However, we argue that it is feasible to borrow ideas from other research areas. For example, in mutation testing, Zhang *et al.* [49] predict whether a mutation can be killed, without executing it. In future work, we plan to adapt their approach to reduce the time of validating generated fixes, which allows constructing more creative fixes.

Extracting more accurate changes. Due to various technical limitations, our extracted changes are not fully accurate. On one side, we build a delta graph for each pair of modified methods. The strategy can lose some changes. For example, a bug fix can modify static code or modifiers. Our underlying tool ignores these changes, although such changes are rare. On the other side, a fix can contain irrelevant changes. For example, when programmes fix a bug, they may find bad smells, and eliminate such smells with refactoring tools. After that, they commit their fixed code, so a commit can contain both fix actions and refactoring actions. As a result, a delta graph can contain irrelevant subgraphs, which leads to lower percents of matched nodes. In future work, we plan to improve our extraction tool. For example, Taneja *et al.* [40] propose an approach that detects refactored code. It is feasible to integrate their tool, so that we can filter irrelevant changes.

6 Related Work

Automatic program repair. Given a buggy program, automatic program repair generates candidate patches and searches for a source code-level patch that fixes the bug and passes all tests [41]. Various approaches are proposed to efficiently search for patches [13, 22, 25, 33, 41]. BugFix [9] uses the apriori algorithm to rank previous fixes. GenProg defines *mutate* and *crossover* operators to generate and evolve patches [41]. RSRepair conducts random search [33]. PAR extracts repair patterns from past bug fixes and prioritizes patch generation accordingly [13]. Liu *et al.* [21] propose an approach that compares bug reports to locate similar past fixes for a new bug. Gao *et al.* [6] repair crash bugs based on existing samples from StackOverflow. Prophet generates most popular patches first, but it trains a machine learning model using past fixes to predict the likelihood of correctness for each generated patch [22]. Rolim *et al.* [36] repair bugs based on known examples. Xiong *et al.* [42] learn how to repair `if`-conditions from code samples and API documents. Chen *et al.* [2] repair bugs with learnt contracts. Chen *et al.* [2] repair bugs with learnt contracts. Le *et al.* [17] synthesize patches based on examples. Saha *et al.* [37] introduce more repair templates and algorithms to rank patches. Le Goues *et al.* demonstrate that GenProg can fix 55 out of 105 bugs [18], while Qi *et al.* argue that only 2 of the 55 patches are actually correct [34]. Le Goues *et al.* [19] prepare a benchmark for the follow-up research in this research field. Recent studies [44, 45] show that better test cases can lead to better synthesized patches. Yang *et al.* [43] show that the suspiciousness-first algorithm is better than the rank-first algorithm in parallel repair and patch diversity. Though many approaches rely on hints from existing patches, there are still fundamental research questions left unanswered, which motivates our empirical study.

Empirical studies on bugs and fixes. Many researchers manually inspect code changes and commit messages to understand bug fixes [10, 46, 47]. Such manual inspection process is too time-consuming to scale, and suffers from human bias. Some other researchers build tools to automatically comprehend code changes, and find that a lot of bug fixes are repetitive. For example, Nguyen *et al.* show that 17%-45% bug fixes are repetitive [32], but their tool focuses on bug fixes involving API calls. Ray *et al.* observe that 11%-16% patches are copy-pasted across different BSD products [35].

However, they only leverage textual similarity to identify similar bug fixes. In real world, nevertheless, many bug fixes may not call any API, and similar bug fixes may be textually different, but syntactically similar. Relevant to automatic program repair, Gabel *et al.* [5] found a lot of syntactic redundancy in Sourceforge projects, but their study does not analyze bug fixes. Barr *et al.* [1] report that 11% bug fixes can be fully reconstituted from existing code. Martinez *et al.* [26] report that at the line granularity, 3% to 17% bug fixes are temporal redundancy. Due the three issues as introduced in Section 1, we believe that our results better reflect the reality of repetitive bug fixes, since we handled all the three issues. By conducting this empirical study to explore how useful past fixes can be given six imaginary ways to reuse past fixes, we are taking a step towards investigating new automatic program repair approaches.

Automatic change comprehension. ChangeDistiller parses code changes between the old and the new versions of a program entity (*i.e.* class, method, and field) [4]. It has been used to automatically comprehend and represent code changes [27, 32, 35]. However, it does not perform static analysis to correlate changes. In contrast, static analysis frameworks, such as Soot and WALA, analyze whole programs without knowing which part is changed. When understanding relationship between code changes, we need a tool to focus static analysis on modified code only. PPA is the state-of-the-art tool for partial code analysis [3], but it does not conduct control or data flow analysis. In order to better understand bug fixes automatically, we use GRAPA to effectively merge all kinds of tools mentioned above.

7 Conclusion

With decades of software development, many bugs fixes accumulate, and such fixes contain valuable knowledge on fixing new bugs. Recently, researchers propose approaches that locate useful fixes and use located fixes to guide the fix process of a new bug. Despite their positive results, their potential is largely unknown, and many questions are still open. In this paper, we conduct an empirical study on millions lines of code to answer the open questions. We summarize our results into twelve findings, and provide our insights for the follow-up research on this topic.

Acknowledgement

We appreciate the anonymous reviewers for their constructive comments. Hao Zhong is sponsored by the National Key Basic Research Program of China (973 program) No. 2015CB352203, the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305. Na Meng is sponsored by the NSF CCF No. 1565827.

References

1. E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proc. ESEC/FSE*, pages 306–317, 2014.

2. L. Chen, Y. Pei, and C. A. Furia. Contract-based program repair without the contracts. In *Proc. ASE*, pages 637–647, 2017.
3. B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. 23rd OOPSLA*, pages 313–328, 2008.
4. B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.
5. M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proc. FSE*, pages 147–156, 2010.
6. Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proc. 30th ASE*, pages 307–318, 2015.
7. G. H. Golub, M. Heath, and G. Wahba. Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, 21(2):215–223, 1979.
8. Y. Higo, A. Ohtani, S. Hayashi, H. Hata, and K. Shinji. Toward reusing code changes. In *Proc. MSR*, pages 372–376, 2015.
9. D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *In Proc. 17th ICPC*, pages 70–79, 2009.
10. G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proc. 33rd PLDI*, 2012.
11. S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proc. ICSE*, pages 266–276, 2014.
12. T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transaction on Software Engineering*, pages 654–670, 2002.
13. D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. 35th ICSE*, pages 802–811, 2013.
14. S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proc. ESEC/FSE*, pages 35–45, 2006.
15. S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proc. 29th ICSE*, pages 489–498, 2007.
16. H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.
17. X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proc. ESEC/FSE*, pages 593–604, 2017.
18. C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. 34th ICSE*, pages 3–13, 2012.
19. C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.
20. G. Liang, Q. Wang, T. Xie, and H. Mei. Inferring project-specific bug patterns for detecting sibling bugs. In *Proc. ESEC/FSE*, pages 565–575, 2013.
21. C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *Proc. 6th ICST*, pages 282–291, 2013.
22. F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proc. 10th ESEC/FSE*, pages 166–178, 2015.
23. F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proc. 43rd POPL*, pages 298–312, 2016.
24. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Proc. 13th ASPLOS*, pages 329–339, 2008.
25. M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2013.
26. M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proc. 36th ICSE*, pages 492–495, 2014.
27. N. Meng, M. Kim, and K. S. McKinley. Sydit: Creating and applying a program transformation from an example. In *Proc. ESEC/FSE*, pages 440–443, 2011.
28. J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.
29. E. W. Myers. Ano(nd) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
30. S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proc. 36th ICSE*, pages 803–813, 2014.

31. H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proc. 28th ASE*, pages 180–190, 2013.
32. T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proc. 32nd ICSE*, pages 315–324, 2010.
33. Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proc. 36th ICSE*, pages 254–265, 2014.
34. Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proc. ISSTA*, pages 24–36, 2015.
35. B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proc. ESEC/FSE*, 2012.
36. R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proc. 39th ICSE*, pages 404–415, 2017.
37. R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: effective object oriented program repair. In *Proc. ASE*, pages 648–659, 2017.
38. L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.
39. M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proc. ICSE, Software Engineering In Practice*, pages 99–108, 2015.
40. K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *Proc. 22nd ASE*, pages 377–380, 2007.
41. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. 31st ICSE*, pages 364–374, 2009.
42. Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *Proc. ICSE*, pages 416–426, 2017.
43. D. Yang, Y. Qi, and X. Mao. An empirical study on the usage of fault localization in automated program repair. In *Proc. ICSME*, pages 504–508, 2017.
44. J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proc. ESEC/FSE*, pages 831–841, 2017.
45. J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering*, pages 1–32, 2017.
46. Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proc. 23rd SOSP*, pages 159–172, 2011.
47. Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. ESEC/FSE*, pages 26–36, 2011.
48. R. Yue, N. Meng, and Q. Wang. A characterization study of repeated bug fixes. In *Proc. ICSME*, pages 422–432, 2017.
49. J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *Proc. ISSTA*, pages 342–353, 2016.
50. H. Zhong and N. Meng. Poster: An empirical study on using hints from past fixes. In *Proc. 39th ICSE*, 2017.
51. H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. 37th ICSE*, pages 913–923, 2015.
52. H. Zhong and X. Wang. Boosting complete-code tool for partial program. In *Proc. ASE*, pages 671–681, 2017.
53. H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.