A Characterization Study of Repeated Bug Fixes

Ruru Yue[1,2] Na Meng[3] Qianxiang Wang[1,2]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[2]Institute of Software, EECS, Peking University, Beijing 100871, China
[3]Computer Science Department, Virginia Tech, Virginia 24060, USA
yueruru@pku.edu.cn, nm8247@cs.vt.edu, wqx@pku.edu.cn

***Abstract*—Programmers always fix bugs when maintaining software. Previous studies showed that developers apply *repeated bug fixes*—similar or identical code changes—to multiple locations. Based on the observation, researchers built tools to identify code locations in need of similar changes, or to suggest similar bug fixes to multiple code fragments. However, some fundamental research questions, such as what are the characteristics of repeated bug fixes, are still unexplored. In this paper, we present a comprehensive empirical study with 341,856 bug fixes from 3 open source projects to investigate repeated fixes in terms of their frequency, edit locations, and semantic meanings. Specifically, we sampled bug reports and retrieved the corresponding fixing patches in version history. Then we chopped patches into smaller fixes (edit fragments). Among all the fixes related to a bug, we identified repeated fixes using clone detection, and put a fix and its repeated ones into one repeated-fix group. With these groups, we characterized the edit locations, and investigated the common bug patterns as well as common fixes.**

Our study on Eclipse JDT, Mozilla Firefox, and LibreOffice shows that (1) 15-20% of bugs involved repeated fixes; (2) 73-92% of repeated-fix groups were applied purely to code clones; and (3) 39% of manually examined groups focused on bugs relevant to additions or deletions of whole `if`-structures. These results deepened our understanding of repeated fixes. They enabled us to assess the effectiveness of existing tools, and will further provide insights for future research directions in automatic software maintenance and program repair.

I. Introduction

Bug fixing is a crucially important task in software maintenance. *Repeated bug fixes* are similar or identical fixes repetitively applied to multiple code locations. Prior studies showed that developers apply repeated bug fixes [16], [28], [32], [34]. For instance, Nguyen et al. found that 17-45% bug fixes were recurring, because the fixes modified API usage in similar ways [28]. Ray et al. conducted a case study of BSD products, and observed that 11-16% patches were copied between OpenBSD, FreeBSD, and NetBSD [34].

Based on the observation that developers apply repetitive bug fixes, various tools have been built to help fix the same bug in multiple locations [29], [26], [24], [23], [14], [21]. For instance, Clever detects and tracks code clones [29]. Each time when a clone is updated to fix a bug, Clever automatically recognizes the clone peers to apply similar edits. SYDIT generalizes program transformation from one code change example, customizes the transformation for user-selected code locations, and then applies the result [24]. LASE generalizes program transformation from two or more similarly changed examples, and leverages the transformation to both find other edit locations, and to suggest similar edits accordingly [23]. PAR exploits 10 manually found common fix patterns to generate candidate program patches, and then tentatively applies each patch to automatically fix bugs [14]. Although repeated fixes have been used to detect or fix bugs, some fundamental characterization questions are still left unanswered, such as when, where, and how repeated fixes are applied. These questions are important for two reasons. First, they help us understand how useful existing repetition-based tools can be, and thus suggest new ways to automatically find or fix bugs. Second, they highlight the common mistakes developers are more likely to make, drawing more developer attention to the pitfalls when building software.

To detect repeated bug fixes, we sampled 1,077, 8,427, and 2,455 bugs in the bug database (i.e., Bugzilla [1]) of Eclipse JDT, Mozilla Firefox, and LibreOffice. In each bug database, every bug has an assigned ID. Similar to prior work [32], [40], we leveraged the bug IDs to retrieve the corresponding *commits* or *patches* in the project's version history, and considered the commits as *bug fixing patches*. We identified in total 19,275 *fixing patches* in this way. To flexibly detect any repetition within a patch or across patches, as with prior work [16], we considered each contiguous block of changed lines in a patch as an individual *bug fix*, and extracted bug fixes from fixing patches accordingly. Next, we leveraged a clone detection tool, CCFinder [13], to identify *repeated bug fixes* for each bug. Finally, we put a fix and its repeated ones into the same *repeated-fix group*.

With all identified repeated-fix groups, we investigated the frequency of repeated fixes, their edit locations, and the semantic meanings. This characterization study addresses the following research questions:

- **What is the frequency of repeated bug fixes?** Overall, a considerable amount of repeated bug fixes were applied in all three projects. At the bug level, 20%, 15%, and 16% of sampled bugs in Eclipse JDT, Mozilla Firefox, and LibreOffice, involved repeated fixes. However, among the different groups of repeated fixes, 57%, 70%, and 48% of groups contained only two repeated fixes, meaning that repeated fixes did not reoccur a lot. 91%, 29%, and 36% of repeated fixes were applied in single patches. Only 30%, 2%, and 6% of repeated-fix groups had fixes repeating among multiple bugs. It indicates that the feasibility of resolving new bugs with past fixes is limited.
- **Where are repeated fixes usually applied?** Repeated

bug fixes were mainly applied to code either in the same package or same file, but not always to code clones. In particular, 86%, 99%, and 91% of repeated-fix groups were purely applied to code within the same package, while 70%, 94%, and 86% of groups were applied to the same file. Both above observations demonstrate the significant spatial locality of repeated fixes. On the other hand, at most 73%, 92%, and 83% of repeated-fix groups were applied purely to code clones. It indicates that repeated fixes are not confined to code clones. When clone detection techniques are not sufficient to reveal all locations in need of similar edits, we may need new approaches that also leverage the spatial locality characteristics to suggest edit locations.

- **What are the common bugs and fix patterns of repeated fixes?** Statistically, 72-82% of repeated fixes required for 3 or more lines of change in Eclipse JDT, Mozilla Firefox and LibreOffice. It means that we need more automatic program repair tools to suggest complicated edits beyond single-line patches. To understand the semantics of repeated bug fixes, we sampled 150 groups of repeated fixes, and manually analyzed the bug patterns and common fixes. Based on our observation, edits to `if`-statements (including statement additions and deletions) and `if`-conditions (including condition additions, deletions, and updates) separately accounted for 39% and 33% of the 150 groups, indicating that developers are more likely to make mistakes when coding `if`-structures. The usual bug-fixing strategies applied to fix such bugs were to add, alter, or delete `if`-conditions or the whole `if`-structures.

These results provide the following three insights for future directions of IDE support, code search, and automatic program repair. First, since many repeated bug fixes are applied within one patch, it is important for IDE to recognize the intent of repetition as early as possible in order to provide helpful coding suggestion in time. Second, when codebase is large and tool response time is critical, code search tools should prioritize search based on spatial locality, since repeated bug fixes are more likely to cooccur within the same package or same file. Third, more advanced automatic repair tools should focus on bugs requiring for additions or deletions of whole `if`-structures and bugs in need of multi-line code changes, since they account for the majority of repeated bug fixes.

II. Concepts

In this section, we define and explain the terminologies used in the paper.

When maintaining software, developers modify code and then commit those changes together with some textual description to version control systems like SVN and Git. In our paper, each code change commit is a ***patch***. The corresponding textual description is called ***commit message***.

Fig. 1 shows a patch example, which represents code changes as textual *diff*s. Each textual diff describes changes applied to one file. When developers modify multiple files, the patch can contain multiple textual diffs. Each textual diff consists of *header information* and a sequence of *hunks* [35]. In the figure, lines 1-4 show the header information, while lines 5-18, and lines 19-34 correspond to two separate hunks.

Each ***hunk*** is a code region, including *context lines* (i.e., unchanged code), and edited lines, such as *deleted lines* (marked with "–") and/or *added lines* (marked with "+"). To facilitate illustration, Fig. 1 uses the black color for context lines, colors deleted lines with **red**, and colors added lines with **bold blue**. A hunk consists of a line delimited by "`@@`", and a code snippet. The `@@`-delimited line describes the starting line and line range of a code change in before- and after-versions, while the code snippet shows edits. A hunk usually contains one ***edit fragment*** (i.e., a contiguous block of changed lines), three context lines above the first changed line, and three context lines below the last changed line, as shown in lines 20-34. However, when two edit fragments are very close to each other and have overlapping context lines, they are merged into one single hunk, as shown in lines 6-18. The edits are noncontiguous (lines 9-10 and line 15), with context lines standing between the two edit fragments.

```
1. diff --git a/org.eclipse.jdt.core/search/org/eclipse/jdt/
   core/search/SearchParticipant.java b/org.eclipse.jdt.core/
   search/org/eclipse/jdt/core/search/SearchParticipant.java
2. index ee39ea4..8d2f5c8 100644
3. --- a/org.eclipse.jdt.core/search/org/eclipse/jdt/core/
   search/SearchParticipant.java
4. +++ b/org.eclipse.jdt.core/search/org/eclipse/jdt/core/
   search/SearchParticipant.java
5. @@ -10,10 +10,13 @@
6.    **********************************************/
7.   package org.eclipse.jdt.core.search;
8.
9. + import java.io.File;
10.+
11.  import org.eclipse.core.resources.IResource;
12.  import org.eclipse.core.resources.IWorkspaceRoot;
13.  import org.eclipse.core.resources.ResourcesPlugin;
14.  import org.eclipse.core.runtime.*;
15.+ import org.eclipse.jdt.internal.core.JavaModel;
16.  import org.eclipse.jdt.internal.core.JavaModelManager;
17.  import org.eclipse.jdt.internal.core.search.indexing.
    IndexManager;
18.
19.@@ -173,8 +176,13 @@
20.  public final void scheduleDocumentIndexing(SearchDocument
    document, IPath indexLocation) {
21.    IPath documentPath = new Path(document.getPath());
22.    IWorkspaceRoot root = ResourcesPlugin.getWorkspace()
    .getRoot();
23.-   IResource resource = root.findMember(documentPath);
24.-   IPath containerPath = resource == null ? documentPath :
    resource.getProject().getFullPath();
25.+   Object file = JavaModel.getTarget(root, documentPath,
    true);
26.+   IPath containerPath = documentPath;
27.+   if (file instanceof IResource) {
28.+     containerPath = ((IResource)file).getProject()
    .getFullPath();
29.+   } else if (file == null) {
30.+     containerPath = documentPath.removeLastSegments(
    documentPath.segmentCount()-1);
31.+   }
32.    IndexManager manager = JavaModelManager
    .getJavaModelManager().getIndexManager();
33.    String osIndexLocation = indexLocation.toOSString();
34.    // TODO (jerome) should not have to create index
    manually, should expose API that recreates index instead
```

Fig. 1: An exemplar patch applied to Eclipse JDT Core.

In our research, a ***fixing patch*** is a patch applied to fix a

bug. Similar to prior work [16], we considered a ***bug fix*** as any edit fragment extracted from a fixing patch without including any context line. We defined ***buggy snippet*** to describe the edit location where a bug fix was applied. Given a bug fix (or an edit fragment), the buggy snippet includes the context lines above the bug fix, the deleted lines in the fix, and the context lines below the fix. A bug fix f is a ***repeated fix*** of another one f' if f and f' are identical or similar. A ***repeated-fix group*** consists of a fix and its repeated ones.

III. Research Questions

In this characterization study, we aim to address the following research questions:

RQ1: *What is the frequency of repeated bug fixes?*

Prior studies show that developers apply repeated bug fixes [16], [28], [32], [34]. However, it is still unclear how repeated fixes distribute across different bugs, different patches, and different hunks. If many repeated fixes can resolve different bugs, we may leverage the observation to maintain a bug fix database which stores all repeated fixes applied in the past, expecting them to help construct valid fixes for future bugs. Le et al. recently created a such tool to conduct history-driven program repair [17].

If many repeated fixes are applied in different patches for the same bug, we can conclude that existing patch-based tools, such as LASE [23], are very useful, because developers forget to apply all repeated fixes at once in many cases. The patch-based tools will significantly reduce later patching burden by inferring program transformation from the initial patch, locating other code in need of similar fixes, and suggesting customized changes. By investigating this research question, we aim to evaluate the effectiveness of existing repetition-based tools, and to identify future directions of tool design.

RQ2: *Where are repeated fixes usually applied?*

When fixes are repetitively applied, we are curious why the repetition happens, and what are the common characteristics of the edit locations. Some tools suggest repeated edits by assuming that fixes always repeat themselves among code clones [29], or among overridden methods in the same class hierarchy [36]. However, no empirical study has been done to validate these assumptions. With this research question, we aim to check them empirically, and may identify some new characteristics if the known assumptions do not always hold.

RQ3: *What are the common bugs and fix patterns of repeated fixes?*

When repeated fixes are applied, we are also curious what are the semantic meanings of repeated fixes, and what are the common problems they try to solve. There are automatic program repair tools built to generate candidate bug fixes based on manually identified common bug fix patterns, such as adding or modifying if-conditions, and altering method parameters [14], [7], [21], [17]. All these fix patterns involve single-line changes. By exploring this research question, we aim to better understand what are the common bugs resolved by repeated fixes, and whether repeated fixes demonstrate any principled way to solve problems.

IV. Methodology

This section describes our approach to identify repeated bug fixes. There are mainly three phases in our approach. Given a subject project, we first leveraged a bug ID-based approach to identify fixing patches (Section IV-A). Next, we filtered hunks and extracted bug fixes from these patches (Section IV-B). Finally, we mined repeated fixes and identified repeated-fix groups (Section IV-C).

A. Identifying Fixing Patches

As with prior work [32], [41], [6], to identify fixing patches in subject projects, we adopted the traditional heuristics [9]. Given a subject project, we first found all reported and resolved bugs in the project's bug database (i.e., Bugzilla [1]). Since each bug has an assigned unique ID, we recorded all IDs of the resolved bugs. Meanwhile, for each commit in the project's version history, we extracted any number mentioned in the commit message, and checked whether the extracted number(s) matched any known bug ID. If so, the commit was considered as a fixing patch for the corresponding bug.

B. Filtering Hunks and Extracting Bug Fixes

From each fixing patch, we extracted hunks to identify bug fixes. We excluded hunks in the files which were less important to program implementation, such as documentation and configuration files. Similar to prior work [25], we also removed hunks that were not inside the implementation of any method or function, such as import statements and comments, since most of these changes did not affect program behaviors.

To identify fixes applied to methods or functions, we took three steps. First, based on the `@@`-delimited line in each hunk, we computed the code range of each bug fix in both old and new versions, namely, ($FRange_o$, $FRange_n$). Second, we used an Abstract Syntax Tree (AST) parser to form a syntax tree for each compilation unit (i.e. source file), and to identify the code range of each method in both old and new versions, namely, ($MRange_o$, $MRange_n$). Finally, for each bug fix, we checked whether its range was inside any known method range. In other words, suppose a compilation unit cu has n methods: $m_1, m_2, \ldots, m_n$. For each bug fix f_i applied to cu, we checked whether there exists $j \in [1, n]$, such that

$$(FRange_o^i \subset MRange_o^j) \wedge (FRange_n^i \subset MRange_n^j) \quad (1)$$

If so, f_i is considered as a fix applied to m_j. Otherwise, if no such method m_j exists, f_i is discarded.

C. Mining Repeated Fixes

With the raw data of bug fixes, we mined for repeated fixes using CCFinder [13]. This mining process takes two steps:

1) Identifying clone regions among fixes: We formatted bug fixes by removing edit operation symbols, such as "+" for added lines and "–" for deleted lines, and then used C-CFinder to identify clone regions. In particular, given multiple formatted fixes, CCFinder first converted each of them to a token sequence by replacing all identifiers of variables, literals, methods, and types with standardized tokens like "$p". It

then detected any common subsequence between the token sequences. As shown in Fig. 2, after the removal of "+" in both fixes, lines 1-3 of f_1 and lines 1-3 of f_2 are converted to the same token sequence, and thus considered clones.

f_1

```
1.+ if (currentLine != null) {
2.+  parseTags(true);
3.+ }
```

f_2

```
1.+ if (line != null) {
2.+  parseOptions(false);
3.+ }
```

Fig. 2: An example of repeated fixes

2) Matching edit operation sequences for cloned regions: We retrieved edit operation symbols for identified clone regions to check whether they matched. In Fig. 2, the edit operation sequence in f_1 consists of three "+" symbols, which perfectly matches the edit operation sequence in f_2. Therefore, the two fixes are considered repetitive. To retrieve as many clones as possible, we configured CCFinder to treat two code snippets as clones if the snippets shared at least 15 tokens.

To better handle the repetitive bug fixes with slight differences, we also defined a similarity metric called *Reflection Ratio* (RR). The intuition is that if two fixes share a large portion of their code changes, the fixes should be considered similar and thus repetitive. Formally, given two fixes f_1 and f_2, if their line numbers are separately n_1 and n_2, and the line number of their overlapping part (the code clone) is $n_{overlap}$, we calculated their separate RR values in the following way:

$$RR_i = \frac{n_{overlap}}{n_i}, \text{ where } i \in [1, 2] \quad (2)$$

If any of the RR values is above a certain threshold, we say that the two fixes are similar. To ensure that we collect enough repeated fixes, the threshold should not be too high. Meanwhile, the majority parts of similar fixes should overlap. Thus, the threshold is set to 60% in our approach.

With the above process, we extracted fixes (edit fragments), identified repeated fixes applied for each bug, and then organized them as repeated-fix groups.

V. Experiments

In this section, we first introduce the subject projects used in our study (Section V-A), and then explain the experiment design and findings for each research question (Section V-B, V-C, and V-D).

A. Subject Projects

We experimented with three open source projects: Eclipse JDT [2], Mozilla Firefox [4], and LibreOffice [3]. Since these projects are popular and widely used, they have well-maintained bug databases and code repositories. The bug reports and commit messages usually have high quality. Therefore, when we identified fixing patches using bug IDs, we almost always retrieved the exact patches corresponding to the bugs. Eclipse JDT consists of five components: Core, Debug, UI, APT, and Text. We chose its main component—Core—for our study. Similarly, our study focused on the mozilla-central component of Mozilla Firefox, and the libreoffice-core component of LibreOffice. To facilitate representation, we still use "Eclipse JDT", "Mozilla Firefox", and "LibreOffice" to label their separate data sets.

TABLE I: Subject projects

Property	Eclipse JDT	Mozilla Firefox	LibreOffice
Programming language	Java	C/C++	C++
LOC of selected component	1,322,334	8,571,183	4,625,297
Resolved period of bugs	2005	2014-2015	2014
# of bugs	1,077	8,427	2,455
# of fixing patches	1,378	10,051	7,846
# of fixes	27,725	183,051	131,080

Table I shows detailed information of our datasets. **Programming language** shows the main language used to implement each project. For example, Eclipse JDT is mainly implemented in Java, while Mozilla Firefox is implemented in C and C++. **LOC** shows the size of each *selected component* under study. For instance, there are 1,322,334 *lines of code* in the Core component of Eclipse JDT.

Since each subject project has a huge number of resolved bugs and commits, we defined a time range to sample a subset of data. All information about the subsets is shown in rows 3-6. **Resolved period of bugs** describes the time period used to sample each project. **# of bugs** illustrates the number of bugs sampled in this way. We obtained 1,077 bugs from Eclipse JDT, 8,427 bugs from Mozilla Firefox, and 2,455 bugs from LibreOffice. With these collected bugs, we used the approach described in Section IV-A to identify corresponding fixing patches. The **# of fixes** row shows the number of fixes extracted from fixing patches.

TABLE II: Refined dataset after filtering

Property	Eclipse JDT	Mozilla Firefox	LibreOffice
# of refined bugs	870	380	1,563
# of refined fixing patches	1,116	569	3,804
# of refined fixes	16,289	3,451	33,057

We further refined the extracted fixes based on the hunks from which the fixes were extracted (as described in Section IV-B). In this way, we obtained 16,289 fixes for Eclipse JDT, 3,451 fixes for Mozilla Firefox, and 33,057 fixes for LibreOffice. As shown in Table II, these fixes separately resolved 870 bugs, 380 bugs, and 1,563 bugs. Comparing Table I and II, we notice that although Mozilla Firefox has more bugs sampled than the other two projects, its number of refined fixes—3,451—is much smaller. The reason is that many fixes in Mozilla Firefox were not applied to function implementation. Instead, they were applied to web scripts, configuration files, and other documents.

With the refined dataset in Table II, we detected repeated fixes using CCFinder, finding 589 repeated-fix groups for Eclipse JDT, 206 groups for Mozilla Firefox, and 1,317 groups for LibreOffice. All these repeated fixes and groups were used in our characterization study.

B. RQ1: What is the frequency of repeated bug fixes?

This research question examines the prevalence of repeated fixes, and further assesses how useful existing repetition-based tools can be when suggesting code changes. In particular, we clustered repeated fixes in various ways, and evaluated the frequency of repeated bug fixes in four dimensions:

D1: How frequently do repeated fixes occur in code repositories?
D2: How dense is fix repetition when it occurs?
D3: How many times do developers usually try to fully apply repeated fixes?
D4: Are there any repeated fixes across bugs?

For D1, we clustered repeated fixes based on the entities (i.e. bugs, patches, and hunks) they were related to. Next, we computed their occurrence rates as the ratios of repeated fix-involved entities to all entities, which are shown below:

$$\text{Bug ratio (Br)} = \frac{\text{\# of bugs resolved by repeated fixes}}{\text{Total \# of bugs}} \quad (3)$$

$$\text{Patch ratio (Pr)} = \frac{\text{\# of patches with repeated fixes}}{\text{Total \# of patches}} \quad (4)$$

$$\text{Hunk ratio (Hr)} = \frac{\text{\# of hunks with repeated fixes}}{\text{Total \# of hunks}} \quad (5)$$

As shown in Table III, there were repeated fixes in all three projects, which corroborates the community's understanding that developers apply repeated fixes. For instance, at bug level, the occurrence rates in Eclipse JDT, Mozilla Firefox and LibreOffice were 20%, 15% and 16%, respectively. However, at patch level, the occurrence rates became 17%, 22%, and 16%. At hunk level, the occurrence rates changed to 12%, 18%, and 21%. Compared with prior studies, our observed numbers are different for two reasons. First, we experimented with a different data set. Second, prior studies [25], [28] measured repetitiveness as the ratio of *the total occurrence of repeated fixes* to *all fixes*, while our computation is based on entities like bugs, patches, and hunks. Our study confirms prior findings from three different perspectives.

TABLE III: Occurrence Rates of Repeated Fixes

	Eclipse JDT	Mozilla Firefox	LibreOffice
# of bugs with repeated fixes	172	58	243
Total # of refined bugs	870	380	1,563
Br	20%	15%	16%
# of patches with repeated fixes	193	123	621
Total # of refined patches	1,116	569	3,804
Pr	17%	22%	16%
# of hunks with repeated fixes	388	192	1,376
Total # of refined hunks	3,221	1,051	6,637
Hr	12%	18%	21%

Finding 1: *A considerable amount of repetitive fixes were found in all three projects. Specifically, 15-20% bugs involved repeated fixes, which corresponded to 16-22% patches, or 12-21% hunks.*

For D2, we counted the instances inside each repeated-fix group, and classified repeated-fix groups based on their instance counts. As shown in Fig. 3, 48-70% of repeated-fix groups contained only 2 fix instances, and 15-19% groups contained 3 repeated fixes. Although in some cases of Eclipse JDT and LibreOffice, a fix repeated more than 30 times to resolve a bug, the cases were really rare.

Some existing tools, such as LASE [23], require developers to provide at least two code change examples to demonstrate a program transformation pattern. Based on the examples, the tool automatically finds other edit locations and suggests similar edits. According to the distribution of repeated-fix groups, we estimate that LASE can help in at most 30-52% cases, because it expects fixes to repeat for at least 3 times. In comparison, tools like SYDIT [24] and LibSync [26], which solely require one edit example to demonstrate the fix pattern, may be more helpful. The reason is when a fix repeats twice, these tools may save half of programming effort by inferring the pattern from one fix, and then suggesting the other fix.

Finding 2: *48-70% of repeated-fix groups contained only 2 fix instances, meaning that for most bugs, repeated fixes did not occur many times.*

For D3, within each repeated-fix group, we identified the patches from which the fixes were extracted, counted the patches, and categorized repeated-fix groups based on the patch counts. As shown in Fig. 4, 73-100% of repeated-fix groups spanned at most 3 patches, meaning that developers usually tried at most 3 times to fully apply all repeated fixes. Especially, 91% of repeated-fix groups in Eclipse JDT have all fixes applied in single patches. It indicates that developers always remembered to apply all repeated fixes in one commit to fix a bug. However, Mozilla Firefox and LibreOffice have many fewer fixes repeating in single patches, which accounted for only 29% and 36% of repeated-fix groups. This observation indicates that for some projects, developers did commit errors of omission, and failed to consistently apply all repeated fixes in one trial.

Researchers proposed approaches to suggest repeated edits based on exemplar edits [23], [24], [26]. However, they have not investigated how to integrate the example-based tools into the software lifecycle. There can be two potential places to introduce automatic edit suggestion: integrated development environment (IDE), and version control system (VCS). When integrated to IDE, edit suggestion tools can monitor developers' edits at runtime, and provide immediate coding assistance once it infers a change pattern and generates some suggestion. When integrated to VCS, edit suggestion tools can analyze each code commit to infer change patterns and suggest edits for next commit. Based on our observations, the former approach of IDE integration seems more promising because the earlier edits are suggested, the more likely they can save developers' editing effort. After the initial patch of repeated fixes, developers may be done with the task, leaving little opportunity for tools to locate and create missing edits.

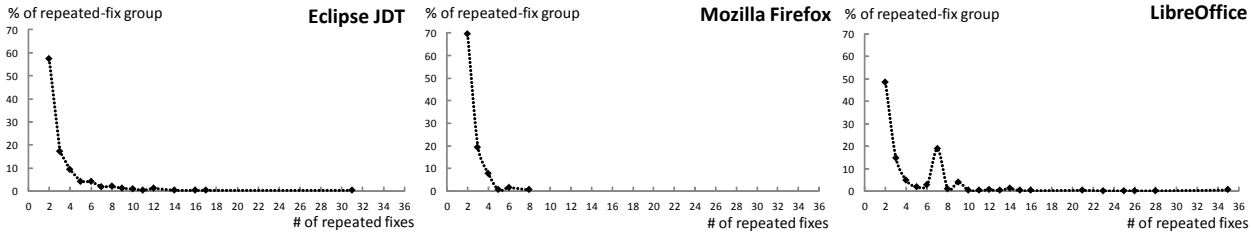

Fig. 3: Distribution of repeated-fix groups based on fix instance counts

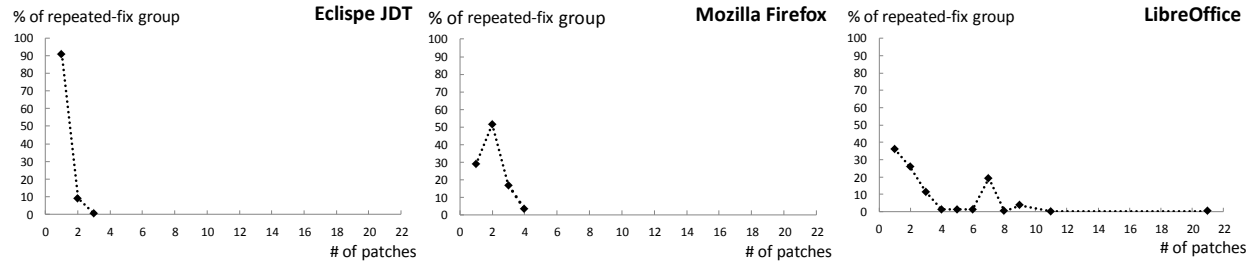

Fig. 4: Distribution of repeated-fix groups based on patch counts

Finding 3: *91%, 29%, and 36% of groups in Eclipse JDT, Mozilla Firefox, and LibreOffice have fixes repeating in single patches. It indicates that coding assistance tools should generate precise edit suggestion as early as possible to effectively help developers.*

For D4, we compared repeated-fix groups to check whether there is any group-level repetition. As each repeated-fix group corresponds to one bug, by identifying repetition between groups, we simulated the experiment to investigate any common fix shared between bugs. As shown in Table IV, we observed group-level repetition in all three projects, although the repetition did not occur very often. In particular, Eclipse JDT has 13% of groups repeated between 2 bugs, meaning that each of these 74 repeated-fix groups could resolve two bugs. In comparison, Mozilla Firefox has only 2% of groups repeated between 2 bugs. It indicates that bugs in Mozilla Firefox are so unique that even if a bug needs repeated fixes, the fixes always have not occurred before.

TABLE IV: Group-level repetition across bugs

Project	Eclipse JDT	Mozilla Firefox	LibreOffice
# of groups	589	206	1,317
# of groups repeated across 2 bugs	74 (13%)	5 (2%)	68 (5%)
# of groups repeated across 3 bugs	51 (9%)	0	12 (1%)
# of groups repeated across 4 or more bugs	42 (8%)	0	5 (0%)

Some automatic program repair tools like Prophet [21], PAR [14], and the tool built by Le et al. [17], generate candidate fixes either based on prior bug fixes mined from open source projects or manually identified bug fix patterns. Our study shows that these tools can help fix bugs to some extent. However, the effectiveness varies from project to project, depending on how unique the bugs are.

Finding 4: *30%, 2%, and 6% of groups in Eclipse JDT, Mozilla Firefox, and LibreOffice have fixes repeated among multiple bugs. It indicates that the feasibility of resolving new bugs with fix patterns derived from past fixes is confined to a small number of bugs.*

C. RQ2: Where are repeated fixes usually applied?

When repeated fixes are applied, we are curious where they usually occur, and what are the common features of their edit locations. Existing tools suggest edit locations for repeated fixes based on similar dependency constraints [23], [24], [36], similar API usage [26], or similar code content (code clones) [19], [29]. For this RQ, we investigated two aspects:

- What is the spatial distribution of repeated fixes?
- Are repeated fixes always applied to code clones?

With the investigation, we aimed to check whether the spatial distribution information can help suggest edit locations, and how effectively clone detection tools can locate buggy code.

To investigate the spatial distribution of repeated fixes, we examined where the fixes were applied for each repeated-fix group. We also checked whether the edited code co-located within the same package, same file, or same method. Then we classified repeated-fix groups accordingly. As shown in Table V, repeated fixes displayed significant spatial locality.

86-99% of groups had all fixes applied in the same package, while 70-94% of groups were applied to the same file. The observation indicates that when codebase is large, and short response time of edit suggestion tools is desired, we can prioritize search scopes based on how probably each scope contains repeated fixes. This investigation shows promising ways to prioritize code search for efficient edit suggestion.

TABLE V: Distribution of repeated-fix groups based on co-location relationship of fixes

Project	Eclipse JDT	Mozilla Firefox	LibreOffice
# of groups	589	206	1,317
# of groups with all fixes in the same package	505 (86%)	203 (99%)	1,200 (91%)
# of groups with all fixes in the same file	412 (70%)	193 (94%)	1,133 (86%)
# of groups with all fixes in the same method	116 (20%)	175 (85%)	912 (69%)

Finding 5: *86-99% of groups consisted of fixes within the same package, and 70-94% of groups were applied to the same file. These distributions demonstrate the strong spatial locality of repeated fixes.*

To estimate how effectively clone detection tools can suggest edit locations for repeated fixes, we started with our repeated-fix groups, and leveraged CCFinder to detect clones in the old version before fixes were applied. Hypothetically, if we observe many repeated fixes occurring purely in code clones, we can conclude that clone detection is always effective to locate bugs for repeated fixes. In order to detect clones surrounding repeated fixes, we first extracted the edit context of each fix as a *buggy snippet*, which includes three context lines before the fix, deleted lines, and three context lines after the fix. Then for each group of repeated fixes, we used CCFinder to find clones between the buggy snippets.

TABLE VI: Distribution of repeated-fix groups whose fixes are purely applied to code clones

Minimum Clone Length (# of tokens)	Eclipse JDT (589 groups)	Mozilla Firefox (206 groups)	LibreOffice (1317 groups)
5	432 (73%)	190 (92%)	1,091 (83%)
10	415 (70%)	190 (92%)	1,078 (82%)
15	411 (70%)	187 (91%)	1,063 (81%)
20	402 (68%)	184 (89%)	1,044 (79%)
25	383 (65%)	173 (84%)	998 (76%)
30	368 (62%)	170 (83%)	967 (73%)
35	352 (60%)	161 (78%)	932 (71%)
40	335 (57%)	158 (77%)	903 (69%)
45	317 (54%)	158 (77%)	879 (67%)
50	305 (52%)	156 (76%)	846 (64%)

Our clone detection results are shown in Table VI. We varied the minimum clone length (MCL) parameter in CCFinder from 5 to 50 tokens, with 5 increment, to comprehensively investigate the distribution of repeated-fix groups. Intuitively, the smaller MCL we set, the more clones we can retrieve. According to the data, if we set MCL to 5 tokens, at most 73-92% of groups have fixes solely applied to clones. If we set MCL to 50 tokens, only 52-76% repeated-fix groups are applied to clones. We manually checked the cases when repeated fixes were applied but no clones were detected in the corresponding buggy snippets. There are mainly two reasons to explain that. First, some buggy snippets contained different numbers of deleted lines. Although the overall fixes (inserted and deleted lines) were similar, the original buggy code was not similar enough to compose clones. Second, some repeated fixes only inserted new code and their context lines were totally different.

Our observations indicate that repeated fixes are not limited to clones, and clone detection cannot always successfully identify all edit locations of repeated fixes. To better search for candidate edit locations, we may need new approaches which incorporate the clone information, the spatial locality, and other possible common characteristics shared between the code snippets requiring for similar edits.

Finding 6: *At most 73-92% of repeated-fix groups were purely applied to code clones, meaning that clone detection alone is not sufficient to identify all edit locations for repeated fixes.*

D. RQ3: What are the common bugs and fix patterns of repeated fixes?

We are curious about the semantic meaning of repeated fixes for two reasons. First, if developers always commit mistakes when writing certain types of code, such code is error-prone, and we may improve existing fault prediction tools [10] to notify developers of potential errors. Second, if developers always apply the same strategy to fix certain types of bugs, it means that there is common fixing practice conducted by developers. If we can leverage such common practice, we may help improve the patches generated by existing automatic program repair tools [21], [14], [37], [18], [22], [7], [27].

Since semantic meanings of fixes are hard to reason about by tools, we manually analyzed fixes. We randomly selected 150 repeated-fix groups, with 50 groups chosen from each project. The manual analysis of these repeated-fix groups was performed by the first author. When analyzing each repeated-fix group, we characterized two things: *bug component*, and *fix pattern*. In most cases, a bug fix always focuses on *one* program syntax component, whether it is a block, a statement, an expression, or a variable. The component manifests the bug in terms of program logic or data usage. We call the program syntax component under fix as *bug component*. It characterizes the bug. When fixes are applied, there are three types of operations involved: *insertion*, *deletion*, and *modification*. Depending on which bug component a fix handles, and how the semantic meaning is modified, we consider the *fix pattern* as updating an `if`-statement, deleting a variable, etc. Notice that each fix pattern characterizes a type of bug fixes. We defined fix patterns by correlating as many lines of changes as possible,

instead of literally reporting changes line-by-line. As shown in Fig. 5, three lines are deleted and seven lines are added. After inspecting the code and correlating changes based on program control and data dependency relationship, we found that the fix aims to modify the assignment to `currFrame` under specific conditions. Therefore, we characterize the fix by identifying the bug component as the `currFrame` assignment statement, and the fix pattern as modifying the assigned value.

In our manual characterization, line changes are correlated either based on the **program dependency relationship** (control or data dependence), and/or the **textual similarity** between inserted and deleted lines. In Fig. 6, the changes are connected based on the textual similarity between insertion and deletion. We therefore summarize the fix as follows: variable `substitutedWildcardBound` is the bug component, and modifying the variable name is the fix pattern. Similarly, in Fig. 7, one inserted line (line 3) is control dependent on the inserted `if`-condition (line 2), while the deleted line (line 1) is similar to that inserted line. Therefore, we summarize the fix to have an `if`-condition bug component, and a fix pattern of adding the condition. Different from above examples, the fix shown in Fig. 8 includes *two* bug components and *two* fix patterns. Since line 1 and 4 are similar, we conclude that `rv`'s assignment is a bug component, and the fix pattern is modifying the assigned value. Meanwhile, lines 1-2 and 4-6 are also similar, and the statements in lines 4-6 control depend on the inserted `if`-condition (line 3). Therefore, the `if`-condition is another bug component, whose corresponding fix pattern is adding the condition.

```
1. -  if (aStart)
2. -    currFrame = aStart->GetNextSibling();
3. -  else
4. +  if (aStart) {
5. +    if (aStart->GetNextSibling())
6. +      currFrame = aStart->GetNextSibling();
7. +    else if (aStart->GetParent()->GetContent()
    ->IsXUL(nsGkAtoms::menugroup))
8. +      currFrame = aStart->GetParent()->GetNextSibling();
9. +  }
10.+  else
```

Fig. 5: The bug component is `currFrame`'s assignment in line 2, and the fix pattern is modifying the assigned value.

```
1. -  this.lowerBound = substitutedWildcardBound;
2. -  if ((substitutedWildcardBound.tagBits
    & HasTypeVariable) == 0)
3. +  this.lowerBound = originalWildcardBound;
4. +  if ((originalWildcardBound.tagBits )
    & HasTypeVariable) == 0
```

Fig. 6: The bug component is `substitutedWildcardBound`, and the fix pattern is modifying the variable name.

```
1. -   CreateAndDispatchEvent(OwnerDoc(),
    NS_LITERAL_STRING("DOMLinkChanged"));
2. +   if (IsInUncomposedDoc())  {
3. +       CreateAndDispatchEvent(OwnerDoc(),
    NS_LITERAL_STRING("DOMLinkChanged"));
4. +  }
```

Fig. 7: The bug component is `if`-condition in line 2, and the fix pattern is adding the condition.

Table VII illustrates our manual inspection results. As shown in the table, for each repeated-fix group, we identified

```
1. -  nsresult rv = RegisterDOMNames();
2. -  NS_ENSURE_SUCCESS(rv, nullptr);
3. +  if (!nsDOMClassInfo::sIsInitialized)  {
4. +      nsresult rv = nsDOMClassInfo::Init();
5. +
6. +      NS_ENSURE_SUCCESS(rv, nullptr);
7. +  }
```

Fig. 8: The bug components are `rv`'s assignment in line 1 and `if`-condition in line 3, and the fix patterns are modifying the assigned value and adding the condition.

the bug component(s) and fix pattern(s), clustered groups accordingly, and counted the number of groups in each cluster for each project. Among 146 out of the 150 groups, there was one bug component and one fix pattern in each fix. However, within the remaining four groups of Mozilla Firefox, each fix contained two bug components and two fix patterns. In Table VII, "-" is used to represent empty entries for simplicity. We ranked bug components based on the number of groups to which they corresponded. From the table, we made the following three observations.

TABLE VII: Manual inspection of 150 repeated-fix groups

Bug Component	Fix Pattern	Eclipse JDT	Mozilla Firefox	LibreOffice
`if`-statement	add `if`-statement	13	14	12
	delete `if`-statement	5	9	5
`if`-condition	modify `if`-condition	4	9	16
	add `if`-condition	9	7	4
	delete `if`-condition	1	-	-
`return`-statement	modify returned value	2	3	3
`assignment` statement	modify assigned value	5	8	3
`case`-branch	add `case`-branch	1	1	2
	delete `case`-branch	1	-	-
`for`-statement	add `for`-statement	1	-	1
`try`-statement	add `try`-statement	2	-	-
`method` `invocation`	modify callee	1	-	-
	modify parameter	1	1	2
variable name	modify variable name	2	2	-
`switch`-statement	add `switch`-statement	1	-	-
`else`-branch	add `else`-branch	1	-	2

Observation 1: `if`-statement and `if`-condition were the most prevalent bug components. Among the identified 11 bug components, `if`-statement and `if`-condition separately accounted for 39% and 33% of all groups. Among the observed 16 fix patterns, adding `if`-statement and modifying `if`-condition were applied the most frequently, separately accounting for 26% and 19%. Among the three fix patterns for `if`-condition, modifying `if`-condition was the most frequently applied one, and developers seldom deleted `if`-conditions. This observation indicates that program repair tools like Prophet [21], Angelix [22], Nopol [7], SemFix [27], and SPR [20] can be useful, because they focus on repairing buggy or missing `if`-conditions. However, we still lack tools to automate `if`-statement additions or deletions, whose edit

operations involve not only changes to `if`-conditions, but also additions or deletions of statements either in the `then`-branches and/or `else`-branches.

Observation 2: Many bugs required control-statement changes or multi-line fixes. By further classifying fix patterns into two categories: control-statement changes and data-value changes, we found the following six components belonging to control-statement changes: `if`-statement, `case`-branch, `for`-statement, `try`-statement, `switch`-statement, and `else`-branch. The other five components correspond to data-value changes. We observed that data-value changes occurred more often than control-statement changes, which was 83 vs. 71. It means that developers are more likely to commit mistakes when defining, checking, or using data. However, they are slightly less likely to make mistakes about control flows.

Existing automatic repair tools like GenProg [18], RSRepair [33], and PAR [14] were built to create single-line patches and correct data values. Angelix can synthesize multi-line patches to correct multiple data values in one fix [22]. Although our manual inspection corroborates that these tools can help fix important data-value bugs, we still see a strong need for tools to create control-statement changes, such as adding or deleting `case`-branches, `try`-statements, etc., because these changes are also important.

Furthermore, we quantified the complexity of repeated bug fixes by computing the LOC of each repeated fix. Surprisingly, only 5%, 1% and 2% of repeated fixes in Eclipse JDT, Mozilla Firefox, and LibreOffice involved single-line additions or deletions. Since a single-line update is often represented as *two lines* in textual diff—one inserted line plus one deleted line, we also counted all two-line changes to assess the upper bound of single-line fix prevalence. At most 17-23% of repeated fixes were covered in this way. In comparison, 77-83% repeated fixes involved 3 or more lines of change. This demonstrates that single-line fixes do not dominate repeated fixes. Program repair tools can be more helpful if they propose multi-line fixes. Le et al. recently created a tool to repair programs by leveraging bug history data [17]. Although the tool could create multi-line fixes, among the 357 bugs in Defects4J [12], the tool can only fix 23 bugs, meaning that more advanced approaches are still needed to automatically fix many bugs.

Observation 3: There were 8 fix patterns applying control-statement changes, and 8 patterns applying data-value changes. Kim et al. once manually inspected human-written patches, and identified the top 8 popular fix patterns that covered almost 30% of all patches they observed [14]. None of their patterns involved control-statement changes. By comparing our pattern set with theirs, we found two overlapping patterns: modifying `if`-condition, and adding `if`-condition. This overlap means programs always contain errors relevant to `if`-statements, which corroborate the previous finding that `if`-condition changes are the most frequently applied bug fixes [31].

We found two reasons to explain the observed pattern divergence. First, Kim et al. focused on the top 8 common fix patterns of more than 60,000 human-written patches, while we reported all observed patterns in 150 repeated-fix groups, which correspond to 435 fixes from 213 patches. Second, their manual inspection was based on groums—a graph-based model for representing object API usage [30]. They discovered similar patches based on the similarity of object API usage. Therefore, if two similar patches involve different APIs, their groums are dissimlar. This limitation can affect Kim et al.'s manual inspection results.

> **Finding 7:** *72% of manually inspected repeated-fix groups focused on bugs in* `if`*-statements or* `if`*-conditions, meaning that* `if`*-statement is the biggest software pitfall. 50% of observed patterns involved control-statement changes, indicating a desperate need for tools to suggest control structure change patches.*

VI. Related Work

This section describes related work on empirical studies of repeated code changes, change recommendation systems, and automatic program repair.

Empirical Studies on Repeated Code Changes. Researchers have observed that developers apply repeated code changes [34], [32], [28], [15], [16]. For instance, Pan et al. found 27 automatically extractable bug fix patterns from 7 open source projects, and observed that `if`-condition changes are the most frequently applied bug fixes [31]. Zhong et al. found that about 3% of bug fixes can be constructed from past fixes based on their similarity or overlap in terms of code names and/or structures [39]. Nguyen et al. found that 17-45% of bug fixes were recurring [28]. They extracted related objects' API usage in modified code before and after each bug fix, and clustered bug fixes based on the graphical representation of API usage modification. Kim et al. observed that on average, 75% of structural changes to mature software involved repeated and consistent changes [15]. Ray et al. applied CCFinder to detect similar patches applied to different BSD products: OpenBSD, FreeBSD, and NetBSD [34]. They observed that developers copied 11-16% patches across the BSD products. Although we used the same tool to mine repeated fixes, our focus is not just to demonstrate the existence of repeated fixes. Instead, compared with all prior studies, we further investigate the edit context where repeated fixes always occur, and what is the semantic meaning of repeated fixes. With such further investigation, we managed to provide relevant guidelines to change recommendation systems and automatic program repair tools.

Change Recommendation Systems. Based on the insight that developers write duplicated code and apply repeated code changes, researchers have proposed various tools to recommend code changes [23], [24], [26], [29], [11], [19]. For instance, CP-Miner [19] and Deckard [11] mine code clones, and further detect copy-paste related bugs based on identifier mappings or context mappings among clones. Clever keeps track of all clone groups in software and monitor for edits applied to any clone. If one clone is detected to be

updated, Clever lists all its clone peers, and recommends relevant changes. LibSync focuses on API usage adaptation edits. It represents API usage adaptation changes as a graph, extracts a feature vector from the graph, and then leverages the feature vector to search for similar code involving similar API usage. SYDIT and LASE take another approach to model and suggest changes [23], [24]. Given an exemplar edit, they leverage program static analysis to identify all dependency relationship between the edit and unchanged code, and extract code from the version before-edit to capture the dependency constraints. The extracted code is then generalized and leveraged to suggest similar edits to other code locations. Instead of proposing another tool, the focus of this paper is to investigate (1) how helpful clone-based tools can be when locating code to apply repeated fixes, and (2) whether there is any missing feature of repeated fixes that can be leveraged to help improve existing tools.

Automatic Program Repair. Researchers have proposed tools to generate candidate patches for certain bugs, and automatically check patch correctness using compilation and testing [21], [33], [7], [14], [18]. For example, GenProg [18] and RSRepair [33] generated candidate patches by replicating, mutating, or deleting code randomly from the existing program. Since the random search usually involves a huge search space, other researchers leveraged repeated fixes to improve the search efficiency. For instance, Kim et al. built PAR to prioritize patch generation for the most popular fix patterns [14], while Long et al. built Prophet to focus patch generation on statistically most possible fix patterns [21]. In terms of patch generation capability, all above tools mainly generate single-line bug fixes. In our study, we roughly estimated how helpful such patches can be to automatically fix bugs. Compared with above search-based and pattern-based tools, Nopol takes a different approach to compare the path conditions of successful runs and failing runs for any condition difference, and then leverages an SMT solver to synthesize the path condition leading to a bug fix [7]. In our study, we confirmed that `if`-conditions are places where developers always make mistakes, and further identify other pitfalls in software development.

VII. Threats to Validity

Our observations are based on subsets of bugs and fixes mined from three open source projects, and may not generalize well to other bugs and projects. In the future, we plan to include more open source projects into our study, and crowdsource the workload of manually checking bugs and fixes. By asking developers to independently observe the bugs and fixes, we will identify the bug components and fix patterns which the majority developers agree on. We will also explore advanced approaches to automatically detect the bug components and fix patterns. For instance, we can use GumTree [8]—a finer-grained source code differencing tool—to precisely extract and represent code changes, and use WALA [5]—a program static analysis framework—to reason about the relationship between changes, and to further infer developers' change intention.

When identifying fixing patches, we reused an existing approach [6] based on the resolved issues in bug databases, assuming that each issue mentions a bug. However, some issues might be actually irrelevant to bugs; instead, they might focus on new requirements or code refactoring. In future, we plan to leverage more advanced approaches [38], [35] to better identify fixing patches.

We used the clone detection tool CCFinder to identify repeated fixes. CCFinder can detect simple clones with different line breaks or different identifier names. However, it cannot detect clones with further variations like statement additions or deletions. Although we leveraged a similarity metric *Reflection Ratio* (RR) to alleviate the limitation by flexibly matching syntactically different repetitive fixes, the threshold setting of RR could be a potential threat to internal validity.

As with prior work [16], we defined a fix as a contiguous block of edits. A limitation of this definition is that developers may apply multiple noncontiguous edits as a whole to fix one bug. However, the goal of this study is not to count the exact number of fixes developers applied for a bug. Instead, we focused on estimating the repetitive coding effort of developers, and characterizing the frequency, context, and meaning of the repetition.

VIII. Conclusion

Prior empirical studies showed that developers applied repeated bug fixes. Many tools were built to suggest repeated fixes to save coding effort and reduce omission errors. In this characterization study, we revisited repeated fixes, and characterized them thoroughly from three perspectives: frequency, edit context, and semantic meaning. Different from prior studies, we not only reported and analyzed numbers, but also correlated every observation with existing tools to investigate how well the tools automated repetitive fixes.

Our study on Eclipse JDT, Mozilla Firefox and LibreOffice shows that 15-20% of bugs involved repeated fixes, but many repeated fixes were applied in the same patch to fix the same bug. At most 73-92% of repeated-fix groups consisted of fixes reoccurring among code clones, but many more repeated fixes demonstrated significant spatial locality. 39% of repeated-fix groups resolved bugs by adding or deleting whole `if`-structures. The observations indicate we need better edit suggestion tools by appropriately incorporating the tools into software lifecycle, by prioritizing code search scopes for efficiency, and by designing more advanced techniques to better reuse past bug fixes to resolve new bugs in future.

Acknowledgment

We thank anonymous reviewers for their thorough and valuable comments on our earlier version of the paper. This work was supported by the National Key Research and Development Program under Grant No. 2016YFB1000801, the National Natural Science Foundation of China under Grant No. 61672045 and 61421091, and NSF Grant No. CCF-1565827.

REFERENCES

[1] Bugzilla. https://bugzilla.mozilla.org.

[2] Eclipse jdt. http://www.eclipse.org/jdt/.

[3] Libraoffice. https://www.libreoffice.org.

[4] Mozilla. https://www.mozilla.org.

[5] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[6] A. Bachmann and A. Bernstein. Software process data quality and characteristics: A historical view on open and closed source projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 2009.

[7] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, 2014.

[8] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[9] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, page 23, 2003.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[11] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[12] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[13] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *TSE*, pages 654–670, 2002.

[14] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *IEEE/ACM International Conference on Software Engineering (to appear)*, 2013.

[15] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ACM/IEEE International Conference on Software Engineering*, pages 309–319, 2009.

[16] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.

[17] X. B. D. Le, D. Lo, and C. L. Goue. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1), January 2012.

[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. pages 289–302, 2004.

[20] F. Long and M. Rinard. Staged program repair with condition synthesis. In *ESEC/FSE '15: Proceedings of the the 10th Joint Meeting of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 166–178, 2015.

[21] F. Long and M. Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 2016.

[22] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ACM/IEEE International Conference on Software Engineering*, pages 691–701, 2016.

[23] N. Meng, M. Kim, and K. McKinley. LASE: Locating and applying systematic edits. In *ICSE*, page 10, 2013.

[24] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[25] 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 *ACM/IEEE International Conference on Automatic Software Engineering*, pages 180–190, 2013.

[26] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. pages 302–321, 2010.

[27] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ACM/IEEE International Conference on Software Engineering*, pages 315–324, 2010.

[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *ASE*, pages 123–134, 2009.

[30] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE '09: Proceedings of the 7th Joint Meeting of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 383–392, New York, NY, USA, 2009. ACM.

[31] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 2009.

[32] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *IEEE Working Conference on Mining Software Repositories*, pages 40–49, 2012.

[33] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE*, 2014.

[34] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *ACM International Symposium on the Foundations of Software Engineering*, page 11 pages, 2012.

[35] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *ACM/IEEE International Conference on Software Engineering*, pages 386–396, 2012.

[36] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 457–466, 2010.

[37] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[38] R. Wu, H. Zhang, S. Kim, and S. C. Cheung. Relink: recovering links between bugs and changes. In *ACM Sigsoft Symposium and the European Conference on Foundations of Software Engineering*, pages 15–25, 2011.

[39] H. Zhong and N. Meng. An empirical study on using hints from past fixes: poster. In *International Conference on Software Engineering Companion*, pages 144–145, 2017.

[40] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, 2015.

[41] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.