An Empirical Study of Multi-Entity Changes in Real Bug Fixes

Ye Wang, Na Meng
Department of Computer Science
Virginia Tech, Blacksburg, VA, USA
{yewang16, nm8247}@vt.edu

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai, China
zhonghao@sjtu.edu.cn

***Abstract*—Prior studies showed that developers applied *repeated bug fixes*—similar or identical code changes—to multiple locations. According to the observation, researchers built tools to automatically generate candidate patches from the repeated bug-fixing patterns. However, all such research focuses on the recurring change patterns within single methods. We are curious whether there are also repeated bug fixes that change multiple program entities (e.g., classes, methods, and fields); and if so, how we can leverage such recurring change patterns to further help developers fix bugs. In this paper, we present a comprehensive empirical study on multi-entity bug fixes in terms of their frequency, composition, and semantic meanings. Specifically for each bug fix, we first used our approach *InterPart* to perform static inter-procedural analysis on partial programs (i.e., the old and new versions of changed Java files), and to extract change dependency graphs (CDGs)—graphs that connect multiple changed entities based on their syntactic dependencies. By extracting common subgraphs from the CDGs of different fixes, we identified the recurring change patterns.**

Our study on Aries, Cassandra, Derby, and Mahout shows that (1) 52-58% of bug fixes involved multi-entity changes; (2) 6 recurring change patterns commonly exist in all projects; and (3) 19-210 entity pairs were repetitively co-changed mainly because the pairs invoked the same methods, accessed the same fields, or contained similar content. These results helped us better understand the gap between the fixes generated by existing automatic program repair (APR) approaches and the real fixes. Our observations will shed light on the follow-up research of automatic program comprehension and modification.

I. Introduction

Bug fixing is important for software maintenance, and developers usually spend a lot of time and effort fixing bugs to improve software quality. Prior studies showed that developers applied repeated bug fixes—textually similar or identical code edits—to multiple locations [40], [52], [75], [73]. For instance, Kim et al. found that 19.3-40.3% of bugs appeared repeatedly [40]. Zhong and Meng observed that more than 50% of code structure changes could be constructed from past fixes [75]. With these observations, researchers proposed various tools to generate bug fixes or suggest customized edits based on the fixes already applied by developers [48], [49], [38], [43]. Specifically, Kim et al. identified 10 common fix patterns in thousands of bug fixes, and developed PAR to automatically create patches from the patterns [38]. Meng et al. built LASE, a program transformation tool that generalizes a code change pattern from multiple similar edits, and leverages the pattern to suggest new edit locations together with a customized edit for each suggested location [49].

Although the bug-fixing patterns are useful for existing automated tools, *the fixes that they focus on are limited to code changes within single methods or edits solving single software faults*. A recent study by Zhong and Su [76] shows that around 80% of real bugs are fixed when multiple program locations are edited together, meaning that the majority of real fixes solve multiple software faults together. However, *it is still unknown among the multi-fault fixes, whether there is any repeated bug-fixing pattern that repetitively applies similar sets of relevant edits to multiple program entities (e.g., classes, methods, and fields)*. If there are such patterns, perhaps researchers can further build automatic tools to generate multi-fault patches that change at least two entities each time.

To deepen our understanding of repeated bug fixes and provide insights for automatic fix generation tools, in this paper, we conduct a comprehensive study on 2,854 bug fixes from 4 projects: Aries [2], Cassandra [3], Derby [4], and Mahout [5]. Our research characterizes multi-entity fixes in terms of their frequency, composition, and semantic meanings. Static analysis tools typically require compilable code for advanced analysis and a recent study shows that only 38% of commits are compilable [64]. Thus, it is challenging to accomplish an empirical study of multi-entity fixes through examination of code commits. To handle this problem, we implemented a tool called *InterPart*, based on prior work [77]. *InterPart* supports interprocedural static analysis on bug-fixing commits even though the commits are not compilable. To analyze a bug fix, we first extended ChangeDistiller [34] to extract all changed entities such as *Deleted Classes (DC)*, *Changed Methods (CM)*, and *Added Fields (AF)*. Next, we used *InterPart* to identify the *syntactic dependency* relationships (e.g., method invocation) between changed entities. If there exists any such dependency between some changed entities, we further built one or more ***Change Dependency Graphs (CDGs)*** to connect the related entities. By extracting common subgraphs from CDGs, we thus revealed the ***recurring change patterns*** that involve multiple entities. Finally, we conducted two case studies to understand (1) why certain types of entities were usually changed together, and (2) why specific entities were co-changed repetitively.

Although some prior empirical studies analyzed the co-

change relationship between source files, program entities, or statements [78], [51], [46], [30], our research is unique for two reasons. First, when developers change multiple code locations in one bug-fixing commit, some of the changes can be totally irrelevant to bug fixing [36]. Instead of blindly assuming all changed entities to be relevant, we used *InterPart* to identify related changes based on the syntactic dependency or referencer-referencee relationship between entities. Consequently, our investigation of recurring change patterns is more precise. Second, our study is not limited to the phenomena shown by automatic analysis, because we conducted case studies to manually inspect co-changed entities for bug fixes, to further explore the rationale underneath the extracted patterns.

In this study, our major research questions and interesting findings are as follows:

- **What is the frequency of the bug fixes involving multiple entity changes?** If the majority of real fixes involve multi-entity changes, it is important to explore any recurring change pattern among such fixes and characterize the co-changed entities. To answer this question, we extended a program differencing tool—ChangeDistiller—to extract eight types of changes involving either classes, methods, or fields. We found that 55%, 52%, 58%, and 52% of the examined bug fixes in Aries, Cassandra, Derby, and Mahout changed multiple entities (Findings 2). Among these multi-entity fixes, we created one or more CDGs for 76%, 74%, 75%, and 66% of fixes (Finding 3). Our observations indicate that developers usually change multiple entities together to fix a bug, and many of such co-applied changes are correlated.
- **What patterns are contained by multi-entity fixes?** By identifying such patterns, we explored new research directions for automatic program repair (APR) and coding suggestion tools. We leveraged an off-the-shelf graph comparison algorithm—VF2 [27]—to compare the CDGs across commits within each project, acquiring six frequent subgraphs or recurring change patterns (Finding 4). The three most frequent patterns change at least one method to fix any bug (Finding 5).
- **Why do programmers make multiple-entity changes, when they fix real bugs?** We conducted two case studies to characterize co-changes. The first study examined 291 fixes that match any of the 3 most frequent patterns to analyze how edits in different entities are relevant. The second study examined 20 entity pairs that were repetitively changed together to explore how the entities in every pair are related to each other. For the recurring change patterns, although we did not see any two fixes that resolve the same bug by applying textually similar edits, we saw that some fixes applied related changes to caller and callee methods for consistent semantic modification (Finding 6). Additionally, we found that 16 of 20 inspected entity pairs had co-changed methods that share field accesses, method invocations, or program content (Finding 7).

Our findings provide three insights for future directions of IDE support, APR, and automatic code change suggestion. First, since many bug fixes may touch multiple entities in one commit, it is important for IDE to explicitly visualize the change type of each touched entity and the connections between those entities. Such visualization will help developers understand the layout of changed entities and the rationale of a bug fix. Second, when developers apply independent changes to multiple related methods, APR approaches may be extended to suggest multi-entity fixes by simultaneously applying the single-method patches generated for individual methods. When developers apply dependent changes to related methods (e.g., modify callee methods to always return `non-null` values and update caller methods to remove `null`-checks), future change suggestion tools can check for such change consistency and help automatically complete fixes. Third, many repetitively co-changed entities have similar field accesses or method invocations, so future change suggestion tools can also suggest edits based on such similarities in addition to the textual similarity. Our project and data are available at https://github.com/yewang16/pattern-finder.

II. Concepts

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

Similar to prior work [60], we use a ***program entity*** to represent a Java class, method, or field. To revise code, developers may add, delete, or change one or more entities. Therefore, we defined a set of ***atomic changes*** or ***changed entities*** to represent any code revision with: *Added Classes* (**AC**), *Deleted Classes* (**DC**), *Added Methods* (**AM**), *Deleted Methods* (**DM**), *Changed Methods* (**CM**), *Added Fields* (**AF**), *Deleted Fields* (**DF**), and *Changed Fields* (**CF**). For instance, if developers create a new class with one field and one method defined in the class, we represent the revision as one **AC**, one **AF**, and one **AM**. As with prior work [76], we refer to a ***bug fix*** as a code revision that repairs a bug. If a bug fix contains multiple atomic changes, we name it a ***multi-entity (bug) fix***.

If a multi-entity fix has relevant atomic changes co-applied, we use one or more ***change dependency graphs (CDGs)*** to represent the related changes. In each CDG, a node represents an atomic change and an edge represents the ***syntactic dependency*** relationship between two entities. We say that an entity E_1 is syntactically dependent on another entity E_2 if (1) E_1 is contained by E_2 (e.g., a field is contained by its declaring class); (2) E_1 overrides E_2 (e.g., a sub-class' method overrides a method in the super class); or (3) E_1 accesses E_2 (e.g., a method accesses a field or invokes another method) [60]. CDG's formal definition is as below:

Definition 1: $CDG =< V, E >$, where V is a set of vertices representing changed entities, and E is a set of directed edges between the vertices $E \subseteq \{V \times V\}$. There is a directed edge from changed entity u to changed entity v, if and only if u is syntactically dependent on v.

Notice that there may be zero, one, or multiple CDGs existing in a code revision. If there is no atomic change in

a revision or the atomic changes are totally irrelevant, we consider that there is no CDG extractable from the revision. CDGs are always extracted from multi-entity fixes.

A ***change pattern (cp)*** is a CDG or a CDG's subgraph, which includes changed entities and the directed connections between entities. Formally,

Definition 2: Given $CDG =< V, E >$, $cp =< V', E' >$, where $V' \subseteq V$ and $E' \subseteq E$. A cp should contain at least two nodes and one edge connecting the nodes.

A ***recurring change pattern (rcp)*** is a change pattern that occurs in at least two program revisions. Formally,

Definition 3: Suppose that the CDGs of code revisions r_1 and r_2 are $GS_1 = \{cdg_{11}, \ldots, cdg_{1m}\}$ and $GS_2 = \{cdg_{21}, \ldots, cdg_{2n}\}$. If a change pattern cp occurs in both cdg_{1i} and cdg_{2j} ($i \in [1, m]$ and $j \in [1, n]$), we say that the change pattern is also an rcp.

Automatic program repair (APR) [70] executes a buggy program P with a test suite T, and leverages bug localization techniques [37], [50], [72], [29] to locate a buggy method. APR then creates candidate patches to fix the bug, and validates patches via compilation and testing until obtaining a patched program that passes T. Different APR approaches generate patches either by randomly mutating code [70], creating edits from the recurring change patterns of past fixes [38], or solving the constraints revealed by passed and failed tests [47]. However, each fix suggested by current APR approaches only modifies a single method.

III. Research Questions

This study aims to address the following research questions:

RQ1: ***What is the frequency of multi-entity bug fixes?*** Prior studies show that developers modified multiple files or entities in one code revision [78], [51], [46], [76], [36], [30]. However, it is still unclear what percentage of bug fixes change multiple entities in single commits, and how such co-changed entities are related to each other. With the understanding of how frequently developers change multiple entities to fix bugs, we can estimate the gap between the fixes output by existing APR approaches and the real fixes, and assess the necessity of exploring recurring change patterns in multi-entity fixes.

RQ2: ***What patterns are contained by multi-entity fixes?*** In bug fixes, certain atomic co-changes may be more closely related than the others, and may occur repetitively in multiple commits of the same or different projects. Such repetitively co-changed entities form the recurring change patterns (*rcp*) in our research. Identifying such patterns can deepen our understanding on how to repair multi-fault programs. Notice that our definition of *rcp* is different from prior work [40], [54], [58], [73], which defined repeated change patterns as similarly added or deleted lines in single methods. In our research, by clustering changes based on their entity-level change types and connections, we identified repetitiveness at the syntactic instead of textual level.

RQ3: ***Why do programmers make multiple-entity changes, when they fix real bugs?*** Several existing tools suggest code changes or bug fixes based on textual or syntactic similarity [48], [49], [38], [43]. We further explored why the above-observed multi-entity change patterns happened and how they revised program semantics. To the best of our knowledge, no prior work identifies or examines such patterns. The answers to this question are important to characterize the scenarios where new tools may suggest nontrivial edits based on semantic similarity to further help developers fix bugs.

IV. Methodology

This section first explains how we created CDGs (Section IV-A) with our newly built tool *InterPart* (Section IV-B). It then describes how we extracted recurring change patterns from CDGs across commits (Section IV-C).

A. CDG Construction

To construct CDG(s) for a given bug fix, we first extracted the atomic changes, and then connected the identified changes based on their syntactic dependence relations.

Step 1: Extracting Changed Entities. Given a bug fix, we extended ChangeDistiller to identify all atomic changes or changed entities. ChangeDistiller is a tree differencing tool for fine-grained source code change extraction. Given two versions of a changed Java file, ChangeDistiller first creates an Abstract Syntax Tree (AST) for each version, and then compares the ASTs to generate an edit script consisting of node insertion(s), deletion(s), update(s), and move(s). ChangeDistiller can report method-level, field-level, and statement-level changes; nevertheless, it does not report class-level changes like *Added Classes* and *Deleted Classes*. Therefore, we modified ChangeDistiller also to detect these two types of changes. Our research leverages the entity-level changes reported by ChangeDistiller.

Step 2: Correlating Changed Entities. To determine whether two changed entities (E_1 and E_2) are syntactically dependent, we check whether there is any containment, overriding, or access relationship between the entities.

- **Containment** checks for any overlap between code regions. If E_1's code region is fully covered by E_2's region, we conclude that E_1 is contained by E_2.
- **Overriding** checks for any polymorphic implementation of methods. If E_1 in a Java class redefines the implementation of E_2—another method defined in the class' super type, we conclude that E_1 overrides E_2.
- **Access** checks for any field or method reference by an edited method. If E_1's implementation refers to E_2, we conclude that E_1 accesses E_2.

With ChangeDistiller's output, we can easily identify the containment relationship. However, to check for the overriding or access relationship, we must resolve the binding information of field and method references in changed entities, and then precisely decide which entity overrides or accesses what other entities. Thus, we built *InterPart* to resolve bindings.

B. Design and Implementation of InterPart

When designing *InterPart*, we went through two comparison phases to decide our implementation strategy.

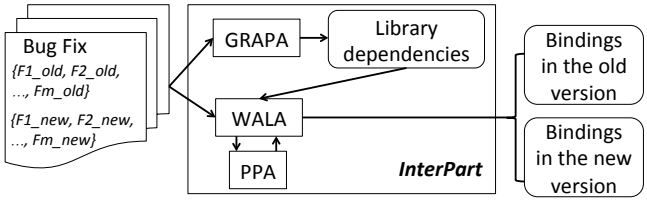

Fig. 1: *InterPart* takes in both the old and new versions of Java files edited for a bug fix, and uses GRAPA, WALA, as well as PPA to resolve bindings in separate versions.

1) Dynamic vs. Static Analysis: It is feasible to resolve bindings of field and method references in a dynamic or static way. With program dynamic analysis, we can execute both the old and new versions of a changed program, and gather the binding information at runtime. However, manually configuring the execution environment to run multiple versions of a program is time-consuming and error-prone. Although a few recent approaches [65], [35] can automatically configure the environment to execute different versions of a program, these approaches only work for certain scenarios. Additionally, dynamic analysis requires high-quality test cases to execute every changed entity for each commit, but this requirement is usually not met in reality. In contrast, static analysis does not require us to execute programs, neither do we need to ensure that a changed entity is covered by one or more test runs. Only Class Hierarchy Analysis (CHA) [31], a well-defined inter-procedural analysis technique, is needed to statically infer the binding information of field or method references. Therefore, we chose static analysis.

2) Complete vs. Incomplete Static Analysis: Inter-procedural program analysis [61] is typically applied to a complete compilable program, including all source files and library dependencies. For example, WALA [22]—a popularly used static analysis framework—requires users to provide all program files and dependencies as inputs for static analysis. Nevertheless, such whole-program analysis may be inapplicable to bug fixes or code revisions. Tufano et al. analyzed the commits of 100 Apache projects, finding that only 38% of commits were compilable [68]. This means that the majority of commits are uncompilable, and we have to manually solve all compilation errors before doing whole-program analysis. In addition, an empirical study [76] shows that a bug fix rarely edits many source files. If we analyze the whole program for a bug fix, the majority of our analysis is wasted on unchanged code. To efficiently resolve bindings in changed files, we chose to conduct inter-procedural analysis on incomplete programs.

There was no ready-to-use tool performing inter-procedural analysis on partial or incomplete programs when we conducted this research, so we built *InterPart* based on WALA, Partial Program Analysis (PPA) [28], and GRAPA [77] (see Fig. 1).

For any bug fix, *InterPart* takes in the old and new versions of edited source files, and mainly relies on WALA to resolve bindings in both versions. When WALA takes in a Java file for analysis, it uses Eclipse ASTParser [21] to parse an AST, and

```
1 public class A {
2   ...
3 + public void foo { ... } //An added method
4 }
5 public class B extends A { ... }//An unchanged class
6 public class C extends B {
7   ...
8  //An added method overriding A.foo()
9 + public void foo { ... }
10 }
```

Fig. 2: A scenario where *InterPart* can miss syntactic dependency information

includes as much binding information as possible in the AST. If a Java file comes from an incomplete program, ASTParser may resolve so few bindings successfully that the resulting AST cannot be processed by WALA. As a solution, we leveraged PPA—a tool to heuristically recover binding information by applying various inference strategies to an Eclipse AST. We extended PPA to correct the inference strategy dealing with local variables and fields. We also extended WALA to (1) take in PPA's AST output, and (2) properly handle `null`-bindings instead of simply throwing runtime errors. In this way, we enabled *InterPart* to conduct inter-procedural analysis on incomplete programs.

To improve WALA's analysis precision, we also need to provide a program's library dependencies as input when analyzing either the old or new version of edited files. Such dependency information is usually not included in a bug fix, and different bug fixes of the same project may depend on different libraries or on the same libraries' different versions. To quickly locate a fix's library dependencies, we used GRAPA [77] to precompute and record the mapping between each fix and the library-version pairs it depends on. Based on such mappings, each time when *InterPart* analyzes a fix, it can query the mappings and locate the relevant library data to load to WALA.

Notice that *InterPart* may miss some syntactic dependencies due to its examination of only changed code—a part of the entire program. As shown in Fig. 2, class `C` extends `B`, while `B` extends `A`. `C` and `A` are separately changed to declare a new method `foo()` (two **AM**s). Ideally, these two changed entities should be related because one overrides the other. However, since `B` is not changed and thus not provided as input, *InterPart* cannot identify the overriding relationship between these **AM**s. Our study may underestimate the dependency relationship between co-applied atomic changes.

C. Recurring Change Pattern Extraction

To investigate what kind of changes are co-applied and how atomic changes are usually related, we extracted recurring change patterns across the CDGs of different commits.

Given two commits' CDGs: $GS_1 = \{cdg_{11}, \ldots, cdg_{1m}\}$ and $GS_2 = \{cdg_{21}, \ldots, cdg_{2n}\}$, we compard every pair of CDGs across commits (e.g., $< cdg_{11}, cdg_{21} >$) in an enumerative way. The comparison of each pair starts with matching individual nodes based on their atomic change labels (e.g., **CM**), and proceeds with edge matching. If two edges

(e.g., e_1 and e_2) have identical source changes, target changes, and edge directions, we consider them as matched.

Based on such node and edge matches, we identified the largest common subgraph between two CDGs with an off-the-shelf subgraph isomorphism algorithm VF2 [27]. Our implementation used JGraphT [19], a Java graph library with a built-in implementation of VF2. Intuitively, VF2 treats the known node matches as initially identified common subgraphs. It then iteratively expands the subgraphs by incrementally adding node or edge matches that do not conflict with the existing match(es). This process continues until no extra match can be added. When VF2 recognized multiple common subgraphs, we recorded the one covering the largest number of nodes, considering it as a recurring change pattern.

V. Empirical Result

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

A. Subject Projects

In our study, we analyzed the bug fixes of four open source projects: Aries, Cassandra, Derby, and Mahout. We chose these projects for three reasons. First, they are from different application domains. Aries [2] supports the OSGi application programming model. Mahout [5] is a library of scalable machine-learning algorithms. Although both Cassandra [3] and Derby [4] are databases, Cassandra is a NoSQL database, while Derby is a relational database. Second, these projects have well-maintained issue tracking systems and version control systems. Developers usually check in high-quality commits with good commit messages to describe the changes. Third, many bug-fixing commits refer to the corresponding bug reports via issue IDs. We downloaded and reused the bug-fixing data of a prior study [76], [10]. In this data set, bug fixes are collected based on the issue IDs of bug reports and commit messages referring to those IDs. An issue ID may correspond to multiple duplicated commits, so we only picked one of those duplicates to avoid bias in the data set.

TABLE I presents more information about these projects. **KLOC** shows the code size of each project. **# of Fixes** shows each project's total number of bug fixes contained by the original data set [76]. Although we tried our best to perform incomplete program analysis on every bug fix, there are still fixes that cannot be processed due to the limitation of PPA or WALA. When either PPA or WALA throws any runtime error for a bug fix, we excluded the fix from our study. Therefore, **# of Fixes Studied** counts the number of fixes in each project that were successfully processed by *InterPart*. All our following investigations are based on these 2,854 fixes. **# of Entity Changes** reports the number of atomic changes we extracted from these fixes.

B. RQ1: What is the frequency of multi-entity bug fixes?

This research question examines the prevalence of multi-entity changes in real fixes, and explores the syntactic dependence relations between co-changed entities.

TABLE I: Subject projects

Property	Aries	Cassandra	Derby	Mahout
KLOC	288	410	1,174	186
# of Fixes	621	3,492	1,591	467
# of Fixes Studied	247	1,515	824	268
# of Entity Changes	1,104	6,739	4,244	1,462

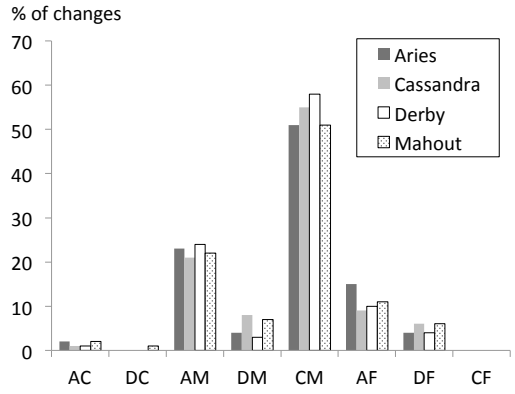

Fig. 3: Change distribution among different change types

As shown in Fig. 3, the extracted changes distribute unevenly among the eight predefined change types. Interestingly, although different developers fix distinct bugs, the relative distributions of different typed changes are very similar across projects. For instance, all the projects have over 50% of atomic changes (i.e., 51-58%) as **CM**s, meaning that developers mainly change existing methods to fix bugs. Over 20% of atomic changes (i.e., 21-24%) are **AM**s. 9-15% of the changes are **AF**s. These observations indicate that in addition to **CM**s, developers also frequently applied **AM**s and **AF**s to fix bugs. Current APR approaches only suggest **CM** patches.

> **Finding 1:** *Similar to the fixes generated by APR approaches, real bug fixes also mainly consist of **CM**s. However, real fixes usually involve a much more diverse set of entities and change types, such as **AM**s and **AF**s.*

We then clustered bug fixes according to the numbers of changed entities they contain, and present the results in Fig. 4. Among the four projects, 42-48% of the fixes contain single changed entities, which means that over 50% of the fixes involve multi-entity changes. Specifically, 15-16% of the fixes include two-entity changes, and 8-9% of the fixes involve three-entity changes. As the number of changed entities increases, the number of fixes decreases in all projects. The maximum number of changed entities appears in Cassandra, where a single fix modifies 240 entities. We manually checked the commit on GitHub [9], and found that it modified 9 files with 4,174 line-additions and 1,048 line-deletions. Existing APR techniques are restricted to proposing single-entity edits.

Zhong and Su conducted a similar experiment on real bug fixes, and found that as the number of repair actions (e.g., inserting, deleting, or modifying a statement) increased, the number of fixes decreased [76]. We observed a similar trend but at the entity instead of statement level.

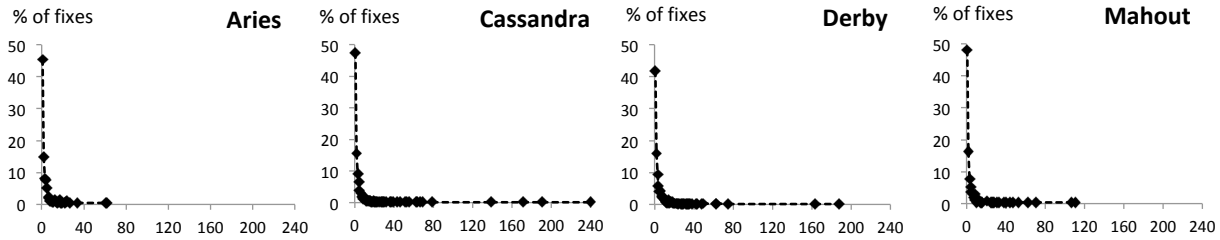

Fig. 4: Bug fix distribution based on the number of included changed entities

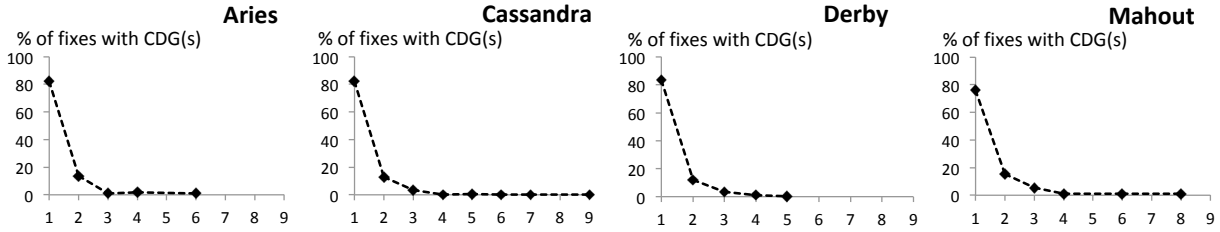

Fig. 5: The distribution of fixes with CDG(s) based on the number of extracted CDGs

Finding 2: *Differing from the fixes generated by APR approaches, over half of the real fixes mainly involve multi-entity instead of single-method changes.*

Among the fixes with multi-entity changes, we further established CDG(s) and clustered these fixes based on the number of CDGs built for them. As shown in TABLE II, **# of Fixes with Multi-Entity Changes** shows the number of fixes that contain at least two atomic changes. **# of Fixes with CDG(s) Extracted** presents the number of fixes for which we built at least one CDG based on the analyzed syntactic dependence relationship between atomic changes. **% of Fixes with CDG(s) Extracted** measures the percentage of fixes with CDG(s) among all multi-entity fixes. We found that 66-76% of the multi-entity fixes contain at least one CDG, which indicates that developers usually change related entities together to fix bugs. Because our incomplete program analysis may miss some dependencies between co-changed entities due to the limited analysis scope (i.e., changed source files only), the actual percentages of related co-changes may be even higher than our measured percentages.

TABLE II: Bug fixes with multi-entity changes

	Aries	Cassandra	Derby	Mahout
# of Fixes with Multi-Entity Changes	135	794	479	139
# of Fixes with CDG(s) Extracted	102	588	357	92
% of Fixes with CDG(s) Extracted	76%	74%	75%	66%

For the fixes containing at least one CDG, we further clustered them based on the number of CDGs extracted. As shown in Fig. 5, 76-83% of such fixes contain single CDGs, which means that the co-changed entities are usually relevant to each other. 12-15% of the fixes contain two CDGs, while 1-5% of the fixes have three CDGs. As the number of CDGs increases, the number of fixes goes down. The fix containing the largest number of CDGs (i.e., 9) is also from CASSANDRA, which involves 19 changed files, 1,367 line-additions, and 239 line-deletions [20]. Prior studies show that developers apply tangled changes or independent groups of changes in one commit [25], [36]. Our observations corroborate that finding.

Current diff tools compare code by text (e.g., UNIX diff [1]), ASTs (e.g., [34] and [33]), and graphs (e.g., [23], [75]). None of these tools visualize any relationship between changes across entities, although developers usually go file by file and context switch to comprehend code changes [66]. Based on our observations of the fixes with multiple related co-changed entities, we think new diff tools based on *InterPart* might better help developers understand changes.

Finding 3: *Among the fixes with multi-entity changes, 66-76% of the fixes contain related changed entities, and 76-83% of such fixes have entities connected in one or more CDGs. This indicates that comparison/recommendation tools that relate co-applied changes will be valuable.*

C. RQ2: What patterns are contained by multi-entity fixes?

This research question explores the recurring co-change patterns in real fixes, and investigates the most frequent patterns among all projects.

As mentioned in Section IV-C, to identify the recurring change patterns, we compared CDGs pair-by-pair across commits and identified the largest common subgraph of each pair (see Section IV-C). Suppose the identified patterns can be represented as $RCP = \{rcp_1, rcp_2, \ldots, rcp_s\}$. To investigate these patterns' frequency, we further matched each pattern $rcp_i (i \in [1, s])$ with all commits' CDGs to check whether a CDG contains one or more subgraphs matching the pattern; if so, the pattern is considered to occur once in the CDG. Notice that if two patterns (e.g., rcp_p and rcp_q) find matches in the same CDG, and one pattern is a subgraph of the other (e.g., rcp_p is contained in rcp_q), we will only count the occurrence of the larger pattern (e.g., rcp_q) to avoid double counting overlapped patterns.

As shown in TABLE III, there are 26, 125, 87, and 24 recurring change patterns separately identified in 4 projects.

TABLE III: Recurring change patterns and their matches

	Aries	Cassandra	Derby	Mahout
# of Patterns	26	125	87	24
# of Fixes Matching the Patterns	97	585	352	87
# of Subgraphs Matching the Patterns	267	1,883	1,270	239

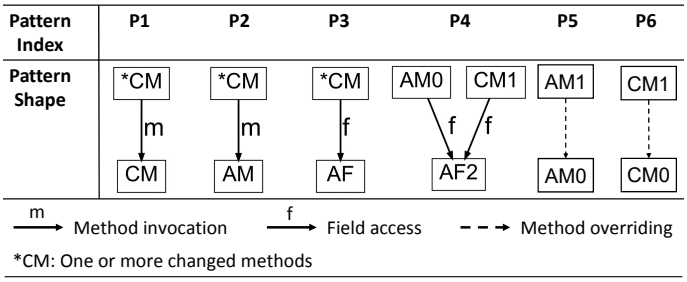

Fig. 6: Six recurring change patterns existing in all projects.

These patterns find matches in 97, 585, 352, and 87 fixes, respectively. If we compare these numbers against the **# of Fixes with CDG(s) Extracted** in TABLE II, it seems that almost every fix with CDG(s) extracted contains at least one subgraph matching a pattern. In total, there are 267, 1,883, 1,270, and 239 matched subgraphs, respectively. There are many more matched subgraphs than matched fixes, indicating that many fixes contain multiple matched subgraphs.

Some APR approaches (e.g., PAR [38]) suggest single-method fixes based on the common patterns of past fixes. The prevalence of multi-entity recurring change patterns indicates that it is promising to extend these APR approaches and generate multi-entity changes from past fixes.

Finding 4: *The fix patterns of multi-entity changes commonly exist in all the investigated projects. This indicates that such patterns may be usable to guide APR approaches and to generate patches changing multiple entities.*

By comparing the extracted patterns from different projects, we further identified six recurring change patterns that commonly exist in all projects, as shown in Fig. 6.

P1: *A callee method experiences a **CM** change, while one or more of its caller methods also experience **CM** changes.* Intuitively, developers usually change one callee method together with the method's caller(s) to fix bugs. This pattern's visual representation contains (1) one **CM*-node to generally represent one or more caller methods, (2) one *CM*-node to represent the callee method, and (3) an *m*-annotated edge to show the caller-callee relationship. The edge's direction presents the invocation direction.

P2: *A callee class experiences an **AM** change, while one or more of its caller methods go through **CM** changes.* Intuitively, when developers add a new method to fix a bug, they usually modify existing method(s) to invoke the new method. This pattern's visual representation is similar to P1's. The **CM*-node still represents one or more changed methods; the *AM*-node represents one added method; and the *m*-annotated edge shows the caller-callee relationship between co-changed entities.

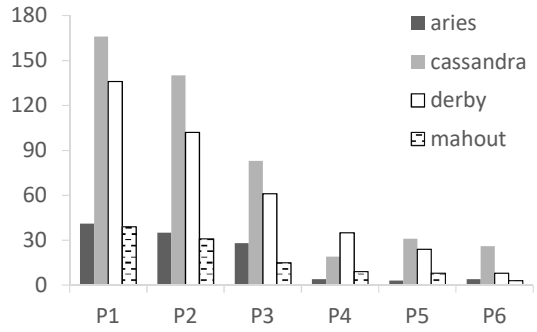

Fig. 7: Bug fixes matching the six frequent patterns

P3: *A field experiences an **AF** change, while one or more methods accessing the field experience **CM** changes.* Intuitively, when developers add a field for bug fixing, they also modify existing method(s) to access the added field. This pattern's representation is similar to above patterns. However, the f-annotated edge shows the field-access relationship between co-changed methods and fields.

P4: *A field experiences an **AF** change, one method undergoes an **AM** change, and another method undergoes a **CM** change.* Intuitively, when developers add a field to fix a bug, they may also add a new method and change a current method to access the field.

P5: *A method experiences an **AM** change (e.g., m_0 is added), while another method overriding m_0 also undergoes an **AM** change.* Intuitively, when developers declare a new method to fix a bug, they may also define a new method to override the declared method. Different from above patterns, the edge has a dashed line to indicate the overriding relationship between methods.

P6: *A method experiences a **CM** change (e.g., m_0 is changed), while another method overriding m_0 also undergoes a **CM** change.* Intuitively, when developers modify a method to fix a bug, they may also modify another method that overrides the method.

To further understand the frequency of the six common patterns, we counted the fixes that match any of these patterns. As shown in Fig 7, the first three patterns (P1-P3) are more frequent than the last three patterns, and *P1-P3 all require to apply at least one CM-change*. Especially, P1 exists in 41, 166, 136, and 39 fixes of separate projects, meaning that 21-30% of the multi-entity fixes co-change the methods with caller-callee relationship. Additionally, we also ranked all patterns mentioned in TABLE III for individual projects based on the number of fixes they match. Interestingly, we found that P1-P3 were the top three most frequent patterns in Aries, Cassandra, and Derby. In Mahout, P1, P2, and P3 were separately ranked as 1^{st}, 2^{nd}, and 4^{th}.

Although existing APR approaches locate single methods to fix bugs, these approaches can be enhanced to (1) locate multiple methods that are correlated via the caller-callee or common field-access dependencies, and (2) generate patches to change those methods simultaneously.

TABLE IV: The scenarios where the most frequent three recurring change patterns occur

Pattern Index	Scenario	% of examined fixes
P1	1. Consistent changes between the callee and callers	23%
	2. Signature change of the callee method	17%
	3. Add, delete, or update the caller-callee relationship	17%
	4. Not closely related changes	43%
P2	1. Add a method for refactoring	28%
	2. Add extra logic or data processing	72%
P3	1. Add a field for refactoring	9%
	2. Partially replace an existing entity	25%
	3. Add extra logic or data processing	66%

Finding 5: *Four out of the six most frequent fix patterns apply multiple CM changes. It indicates that existing APR approaches can be extensible to generate multi-entity fixes by modifying several methods that call the same changed method or access the same added field.*

D. RQ3: Why do programmers make multiple-entity changes, when they fix real bugs?

This question checks for the content and context of recurring change patterns. We conducted two case studies to analyze some fixes matching the patterns. In the first case study, we manually examined 291 fixes that contained any of P1-P3, and explored why and how the multi-entity changes were applied. In the second case study, we further extracted the entity pairs that were repetitively co-changed in version history, and inspected 20 of such pairs to investigate any characteristics.

a) Case Study I—Exploration of Recurring Change Patterns: To investigate the concrete scenarios where the most frequent three recurring change patterns (P1, P2, and P3) occur, we examined 100 bug fixes for P1 by sampling 25 matched subgraphs in every project. Similarly, we inspected another 100 sampled bug fixes for P2. We sampled 91 bug fixes for P3 inspection, because Mahout only has 16 subgraphs matching P3. All our observations listed in TABLE IV are based on these 291 sampled fixes.

- **Scenarios for P1 (*CM$\xrightarrow{m}$CM)**. We identified four scenarios when both callee and caller methods experienced **CM** changes. First, in 23% of the cases, developers applied *consistent changes* to these methods. Among these cases, sometimes caller methods were adapted to the changes in the callee methods. For instance, when a callee method was changed to always return `non-null` values, caller methods were also changed to remove any `null`-check for those returned values. In other cases, both the caller and callee methods were modified to access the same field or invoke the same method. Additionally, in 17% of the 100 cases, developers changed caller methods because the callees' method signatures were changed. In another 17% of the cases, developers changed the implementation logic of callees, and also modified callers to add or delete invocations to the callees, or to change the logic before invoking callees. In 43% of the cases, it seems that there is no obvious relationship between the edits in co-changed methods. Developers might apply independent changes to fix multiple faults simultaneously.
- **Scenarios for P2 (*CM$\xrightarrow{m}$AM)**. We found two scenarios when a callee method was added, and one or more current methods were changed to invoke the added method. First, in 28% cases, developers added the new method for refactoring purposes such as extracting a method or moving a method between classes. Correspondingly, the caller methods were changed to replace some code with the callee method invocation. Second, in 72% cases, developers added a method to implement new logic, and changed current methods to invoke the added method.
- **Scenarios for P3 (*CM$\xrightarrow{f}$AF)**. We found three scenarios when a field was added, and one or more current methods were changed to access the added field. First, in 9% of the cases, developers added a field for refactoring, such as declaring a field for an expression or a constant. Correspondingly, methods were changed to replace some expressions or constants with the field access. Second, in 25% of the cases, developers applied changes to enhance existing features. They declared a new field to partially replace the usage of an existing field or method invocation in certain methods. Third, in 66% of the cases, developers applied changes to add new features, such as adding extra condition checks or data processing.

Prior user studies revealed that developers refactored code when fixing bugs [39], [63]. Our analysis of P2 and P3 corroborate this finding with empirical evidences from real fixes. Among all examined fixes, *we did not see any two identical or textually similar fixes*. It means that multi-entity fixes seldom overlap in textual content. Even though an APR tool like PAR [38] can fix new bugs based on past fixes, extra domain knowledge is required for such tools to generate a correct multi-entity fix completely. For P1, we saw both seemingly irrelevant changes and consistent changes applied to caller and callee methods. Therefore, in addition to extending current APR approaches to simultaneously change related methods and fix multi-fault programs, future research may also automatically check for the change consistency across entities or suggest consistent changes.

Finding 6: *Among P1-P3, we did not see any identical fixes. It means that APR approaches are unlikely to independently suggest a correct multi-entity fix purely based on past fixes, although it is still feasible for new tools to help complete developers' fixes.*

b) Case Study II—Examination of Repetitively Co-Changed Entities: To explore whether there is any characteristic of entities that induces co-changes, among the fixes with CDG(s) extracted, we automatically extracted the entity pairs that were co-changed in at least two fixes. As shown in TABLE V, the majority of repetitively co-changed pairs are methods. Suppose that a pair of entities (e.g., $< E_1, E_2 >$) are co-changed in two fixes (e.g., f_1, f_2). If the change type

of each entity remains the same in both fixes, we conclude that the pair's change types are preserved. Among the 344 pairs, 260 pairs have their change types preserved across fixes. We randomly selected 5 entity pairs from each project whose change types were preserved, obtaining 20 pairs for further manual inspection. All these entity pairs are **CM** changes. As some pairs of entities were co-changed in 2, 3, or 6 fixes; we checked 54 fixes to characterize the 20 pairs (see TABLE VI).

TABLE V: The repetitively co-changed entity pairs

Co-Changed Entities	Aries	Cassandra	Derby	Mahout
Two fields	3	6	0	0
Two methods	16	203	93	22
One class and one field	0	1	0	0

TABLE VI: Characteristics of repetitively co-changed pairs

Characteristics	# of Pairs	Similar statement change?
Similar statements	7	✓
Relevant usage of fields	5	✗
Commonly invoked methods	4	✓
Unknown	4	✗

Seven method pairs contain similar statements, and thus the two methods of each pair were usually changed similarly together. For example, `EmbedResultSet.insertRow()` and `EmbedResultSet.deleteRow()` were co-changed in six examined fixes of Derby. Four of these fixes contain identical statement-level changes [14], [16], [13], [15], while one fix has partially identical changes [12].

Five pairs contain similar or relevant usage of fields. For instance, `Activator.start(...)` and `Activator.stop(...)` were found in three fixes [7], [8], [6]. Both methods accessed the same fields. Although the program context was different and statement-level changes were usually dissimilar, both methods were repetitively changed together to access the same newly added fields (e.g., `registrations`, `ofBuilder`, and `environmentUnaugmentors`).

Four pairs invoke the same sets of methods. For example, `DeleteResultSet.setup()` and `DeleteVTIResultSet.openCore()` were observed to co-change three times [18], [11], [17]. Both methods invoked the same methods. Although their method bodies were different, the two methods were repetitively changed identically to update same method invocations (e.g., `new TemporaryRowHolderImpl(...)`).

Four method pairs were co-changed for some unknown reasons. We could not find any commonality in the program context or textual edits.

Several tools were built to suggest similar statement-level edits to similar code snippets [55], [48], [49]. Our observations demonstrate that such tools can effectively help developers precisely locate and apply edits. More importantly, we found that similarly used fields or methods can also indicate what methods should be co-changed or even what edits should be similarly applied. However, we have not seen any research done to automate such suggestion.

Finding 7: *The repetitively co-changed entities usually share common characteristics like similar content, relevant field usage, or identical method invocations, among which the similar usage of fields or methods has not been leveraged to automatically complete developers' fixes.*

VI. Threats to Validity

Threat to External Validity: Our observations are based on the empirical results from four open source projects. Although we analyzed thousands of bug fixes, our observations may not generalize to other projects. Tian et al. [67] and Wu et al. [71] propose approaches to identify bug fixes even when commit messages do not refer to issue IDs. In the future, we plan to collect more projects with the above approaches.

Threat to Construct Validity: Our manual inspection explores the rationale behind recurring change patterns and repetitively co-changed entities. Although this approach reveals some deep insights, it is not scalable and may be subject to human bias. In the future, we will also design new approaches to automate the inspection process and to uncover more insights in an efficient way.

Threats to Internal Validity: *InterPart* may miss some dependency relations between co-applied changes, due to the limited analysis scope and static analysis nature. For example, *InterPart* only analyzes edited source files, while some real fixes also involve modifications to configurations or metadata (e.g., code annotations and XML deployment descriptors). In the future, we will also extend *InterPart* to correlate the edits in source code and other types of files. The actual relevant co-changes may be more frequent, showing a stronger need for future tools to facilitate change dependency comprehension.

As with prior work [60], we leveraged the syntactic dependence relationship to link relevant changes. It is possible that developers may correlate changes based on other kinds of dependencies. In the future, we also plan to conduct a user study with developers to investigate (1) how syntactic dependence relationship is helpful, and (2) what other relations developers consider when grouping relevant changes. Such investigation will help us better design program differencing tools to facilitate change comprehension.

VII. Related Work

The related work includes empirical studies on code changes, change impact analysis, and APR.

A. Empirical Studies on Code Changes

Researchers showed that developers applied repeated code changes [40], [54], [58], [73]. For instance, Yue et al. found that 15-20% of bugs were fixed with code changes involving repetitive edits [73]. Nguyen et al. reported that 17%-45% of bug fixes were recurring [54]. However, these studies treated co-changed entities independently while ignoring any relationship between those entities. They identified recurring change patterns either based on the string similarity of edited lines or API usage's exact match. In comparison, *InterPart* links changed entities based on their syntactic dependencies,

and reveals the recurring co-change relationship between related entities. For our case studies, we further examined reasons to explain the co-changed related entities. Our study complements all prior work.

Other researchers observed and predicated the co-change relationship between multiple program files or entities in one commit [78], [26], [51], [46], [30]. For instance, Mcintosh et al. mined co-changed source files and built files from version control repositories to characterize the co-change relationship between both types of files [46]. Zimmerman et al. mined co-changed rules between software entities, and then predicted changes based on the mined rules [78]. Herzig et al. found that a non-trivial portion of program commits contained more than one bug fix, feature, or refactoring. [36]. Barnett et al. correlated one program commit's edited lines based on the def-use relationship of types, fields, methods, and local variables; and reported that a significant fraction of commits included multiple groups of correlated changes [25]. Our findings are largely consistent with the above studies but more accurate, as we performed more advanced analysis. More importantly, we manually inspected co-changed entities to understand why they were co-changed in certain ways, and how to automatically locate or apply part of the edits.

B. Change Impact Analysis

Change impact analysis identifies the potential consequences of a change, or estimates what needs to be modified to accomplish a change [24]. A collection of dynamic or static techniques were built to determine the effects of source code modifications. These techniques help developers decide how to augment test suites, which regression tests to select, and what program changes introduce bugs [62], [56], [57], [60], [74], [64]. For instance, Chianti analyzes two versions of a program and decomposes their difference into atomic changes such as CM and AF [60]. By building a call graph for each test case, Chianti decides (1) which test cases are potentially affected by a given set of atomic changes; and (2) what atomic changes may affect the behavior of a particular test case.

Similar to Chianti, we also defined a set of atomic changes roughly at the method level, and analyzed change dependencies to construct CDGs. However, our research is different in two ways. First, our goal is not to explore how code changes affect the unchanged portion's program behaviors. Instead, we explored the recurring patterns of co-applied atomic changes, and further investigated the rationale behind those co-changes. Second, Chianti uses dynamic analysis to reason about change impacts, and requires the full coverage of test cases on edited code; while our approach conducts static analysis on partial code to flexibly characterize change dependencies, without being limited by any test case's coverage on edited code.

C. Automatic Program Repair (APR)

APR tools generate candidate patches for a given buggy program, and automatically check the correctness of each patch using compilation and testing [42], [41], [38], [53], [69], [59], [32], [44], [45], [47]. Specifically, GenProg generates candidate patches by replicating, mutating, or deleting single-line code randomly in the existing program [42]. Prophet trains a machine learning model with successfully applied human patches obtained from open source repositories, and then generates a space of candidate patches to fix bugs [45]. Genesis generates program transformations (e.g., wrapping existing code with a `try-catch` construct) from past fixes, and then uses the transformations to guide automatic patch creation [43]. Angelix leverages constraint solving to synthesize multi-line patches to correct multiple if-condition checks in one fix [47]. To the best of our knowledge, all the above approaches modify only single method bodies.

Our study quantitatively investigates the distribution of real bug fixes across different program entities and change types. It not only shows the significant gap between APR fixes and real fixes, but also identifies potential ways to close the gap by enhancing APR approaches for more general program repair and by proposing new tools for coding completion.

VIII. Conclusion

Our work was intended to exploring the frequency, content, and semantic meanings of multi-entity bug fixes. We built *InterPart* that supports inter-procedural analysis on changed source files, and precisely identifies the syntactic dependencies between co-changed entities. With *InterPart*, we analyzed thousands of bug fixes and revealed various novel findings.

- Multi-entity fixes are frequently applied by developers. In 66-76% of such fixes, the co-changed entities in a bug fix are closely related to each other via syntactic dependencies, which indicates a strong need for program diff tools that visualize the relationship.
- There are three major recurring patterns that frequently connect relevant co-changed entities. As multiple entities are usually co-changed in bug fixes, it is worthwhile exploring how to synthesize fixes that can change multiple syntactically related entities together.
- Although a multi-entity fix is never identical to other fixes, the fix may apply similar or divergent edits to the entities with similar textual content, field usage, or method invocations. This indicates that future tools can suggest missing changes based on the similar content of edited locations, similar usage of fields, or common method invocations between software entities.

We provided actionable advice based on our observations. Our future work is on building automatic tools for change comprehension, program repair, and fix completion.

Acknowledgment

We thank anonymous reviewers for their valuable comments on our earlier version of the paper. This work was supported by NSF Grant CCF-1565827, ONR Grant N00014-17-1-2498, National 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.

REFERENCES

[1] 5 UNIX diff Command Examples of How to Compare Two Text Files. http://www.livefirelabs.com/unix_commands/5-unix-diff-command-examples-of-how-to-compare-two-text-files.htm.

[2] Apache Aries. http://aries.apache.org.

[3] apache/cassandra. https://github.com/apache/cassandra.

[4] apache/derby. https://github.com/apache/derby.

[5] apache/mahout. https://github.com/apache/mahout.

[6] ARIES-1117: Cleanup Aries JNDI from JNDI Map of provider URLs via 'unaugmentation'. https://github.com/apache/aries/commit/d90eeba.

[7] ARIES-311: Implement the JNDI Context Manager service. https://github.com/apache/aries/commit/4bf53f2.

[8] ARIES-318: Be consistent and display nicer error messages if InitialContextFactoryBuilder or ObjectFactoryBuilder are already set. https://github.com/apache/aries/commit/99452a8.

[9] Backport MultiSliceRequest. https://github.com/apache/cassandra/commit/7f63b1f958f0f502f04d24f8d820d29bd484786d.

[10] The collected bug fixes. https://github.com/monperrus/real-bug-fixes-icse-2015.

[11] DERBY-1112: TemporaryRowHolderResultSet is breaking the contract of getActivation. https://github.com/apache/derby/commit/81b9853.

[12] DERBY-1767: insertRow(), updateRow() and deleteRow() cannot handle table names and column names containing double quotes. https://github.com/apache/derby/commit/69d1cb8.

[13] Derby-2566: Outofmemory/sanity-assert failed when updating database. https://github.com/apache/derby/commit/c934682.

[14] DERBY-3261 and part of DERBY-3037. https://github.com/apache/derby/commit/7e374135c0e4e42b2dc6406b54cd015f34d6ca44.

[15] DERBY-3897 SQLSessionContext not correctly initialized in some non-method call nested contexts. https://github.com/apache/derby/commit/e3883f5.

[16] DERBY-4198 When using the FOR UPDATE OF clause with SUR. https://github.com/apache/derby/commit/9f0f05d5e958f9425c0e1366e8e6b844e03c6e39.

[17] DERBY-4488: Nullpointer when performing INSERT INTO. https://github.com/apache/derby/commit/300bbeb.

[18] DERBY-4610: Error attempting delete with cascade and triggers. https://github.com/apache/derby/commit/c69c8b01426926245124eaf21bcc73e1c3c03950.

[19] JGraphT. http://jgrapht.org.

[20] update to thrift trunk and rename jar after the svn revision (806014).... https://github.com/apache/cassandra/commit/0f56a255539e884721afe5a31ec605e935774b71#diff-ca014f74bb639f3e5f3ce32f0e52232d.

[21] Use JDT ASTParser to Parse Single .java files. http://www.programcreek.com/2011/11/use-jdt-astparser-to-parse-java-file/.

[22] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page, 2018.

[23] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. ASE*, pages 2–13, 2004.

[24] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[25] M. Barnett, C. Bird, J. a. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 134–144, Piscataway, NJ, USA, 2015. IEEE Press.

[26] D. Beyer and A. Noack. Mining co-change clusters from version repositories. Technical report, Ecole Polytechnique Federale de Lausanne, 2005.

[27] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct 2004.

[28] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. ACM.

[29] T. Dao, L. Zhang, and N. Meng. How does execution information help with information-retrieval based bug localization? In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 241–250, Piscataway, NJ, USA, 2017. IEEE Press.

[30] M. C. de Oliveira, R. Bonifácio, G. N. Ramos, and M. Ribeiro. Unveiling and reasoning about co-change dependencies. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 25–36, New York, NY, USA, 2016. ACM.

[31] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.

[32] 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*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[33] 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.

[34] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[35] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proc. International Conference on Software Engineering*, 2018.

[36] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, Apr 2016.

[37] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[38] 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.

[39] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Trans. Softw. Eng.*, 40(7):633–649, July 2014.

[40] 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.

[41] 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 *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[42] 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.

[43] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, 2017. ACM.

[44] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[45] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 51, pages 298–312, New York, NY, USA, Jan. 2016. ACM.

[46] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan. Mining co-change information to understand when build changes are necessary. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 241–250, Sept 2014.

[47] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[48] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 329–342, New York, NY, USA, 2011. ACM.

[49] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press.

[50] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, Aug. 2011.

[51] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813, New York, NY, USA, 2014. ACM.

[52] 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.

[53] 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*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[54] 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.

[55] 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.

[56] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5):128–137, Sept. 2003.

[57] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings. 26th International Conference on Software Engineering*, pages 491–500, May 2004.

[58] 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.

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

[60] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[61] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[62] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 46–53, New York, NY, USA, 2001. ACM.

[63] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016.

[64] X. Sun, B. Li, H. Leung, B. Li, and J. Zhu. Static change impact analysis techniques. *J. Syst. Softw.*, 109(C):137–149, Nov. 2015.

[65] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *Proc. 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 366–369, 2012.

[66] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM.

[67] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proc. 34th ICSE*, pages 386–396, 2012.

[68] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[69] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 356–366, Piscataway, NJ, USA, 2013. IEEE Press.

[70] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[71] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[72] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200, Sept 2014.

[73] R. Yue, N. Meng, and Q. Wang. A characterization study of repeated bug fixes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[74] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32, Sept 2011.

[75] H. Zhong and N. Meng. Towards reusing hints from past fixes -an exploratory study on thousands of real samples. *Empirical Software Engineering*, 2018.

[76] 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.

[77] H. Zhong and X. Wang. Boosting complete-code tool for partial program. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 671–681, Oct 2017.

[78] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.