A Study of Multi-Location Bug Patches

Anonymous Authors

***Abstract*—Automatic program repair is a promising approach for reducing the cost of quality assurance practices and faulty software. To date, most techniques proposed for test-driven automatic repair have succeeded primarily on bugs that benefit from short, single-location patches. Techniques that successfully generate multi-location patches often do so in an alternative, single-edit way, or by targeting particular multi-location bug patterns. Empirical studies of real-world similarly tend to focus on the patterns exhibited by single-location bug patches, and have not examined repairability of multi-location patches in detail. We present a comprehensive empirical analysis of multi-location patches for bugs in open source Java programs, focusing on static and dynamic properties that define the repair search space for a given bug. This analysis focuses on the key challenges of the dynamic program repair problem: the *mutations and fix code* used to repair bugs in multiple locations; the *fault locations* and their relationships; and the *objective function*, and in particular how and to what degree test cases can be used (or not) to identify partial repairs. We identify key takeaways and challenges, with implications for future work in expressive, multi-location bug repair.**

***Index Terms*—software bugs, program repair**

I. Introduction

Buggy software has a significant economic cost [**?**], and software failures are estimated to have affected half of the world's population [**?**]. This motivates research in techniques to automatically find and fix bugs. Some of these techniques have begun to be integrated which have become integrated into real-world development practices [**?**], [**?**].[1]

A significant class of program repair techniques in both update this citation list to include stuff published since 2017, plz research [**?**], [**?**], [**?**], [**?**] and practice [**?**] use test cases to guide patch construction. At a high level, these techniques use test cases to localize a defect — identified by at least one failing test — and to validate which (if any) of the generated patch candidates lead the program to pass all tests. Patches are constructed in a variety of ways, ranging from heuristic, syntactic as above, let's join the roaring 20s program manipulation [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], to specially adapted program synthesis techniques [**?**], [**?**], [**?**], [**?**], [**?**]. These techniques have successfully repaired real, meaningful defects in large, complex programs [**?**], [**?**], [**?**], [**?**].

Practically speaking, though, these techniques are typically limited in the types and complexity of defects they can repair. Often this is by design: techniques may limit the repair search space to single-location patches for tractability [**?**], [**?**], [**?**], while others only target certain classes of bugs [**?**], [**?**], [**?**], [**?**]. However, even techniques that can in principle generate repairs with multi-location patches typically don't [**?**]. Meanwhile, many real-world bugs require multi-edit patches [**?**], [**?**], leaving a large propertion of them unrepairable by modern research techniques in program repair.

This is a difficult problem. A key tension in the design of an automated repair technique is the balance between giving users confidence in patch correctness by maximizing its subjective quality while managing a trivially-infinite search space. This space is typically parameterized along several axes: (1) the *fault space*: potential program locations to be modified, (2) the *mutation space*: which modifications may be applied at a location, and (3) the *fix space*: code that may be instantiated for a specific mutation. For example, a repair technique might identify the location of a null-pointer dereference (exploring the fault space), decide to insert new code (mutation space), and synthesize code to initialize the pointer (fix space). Many dynamic repair techniques can be compared in terms of their choices along each of these axes, and traversal strategies have first-order implications for scalability and for the type and quality of patches produced.

Multi-location repair poses distinct challenges for each of these repair axes. Spectrum-based fault localization [**?**], the most prevalent class of fault localization used in program repair, does not specifically identify sets of potentially-related locations; indeed, the evaluation of most fault localization techniques typicaly assumes that a bug is localized if any one buggy line is identified [**?**]. Some bugs *are* repaired in multiple locations using very similar code [**?**], [**?**], informing novel techniques that constrain the *fix space* of possible multi-location repairs accordingly. But, one key question for applicability of these types of techniques is how prevalent such bugs are, relative to bugs that require multiple coordinating (but ultimately different) edits. Test cases are used to evaluate candidate repairs in dynamic repair techniques, but, anecdotally, may not be effective at identifying partial repairs in a multi-location context [**?**]. Although multi-location repair has been discussed in the context of other analyses that study bug fix characteristics in general [**?**] as well as for repair applicability specifically [**?**], [**?**], to the best of our knowledge there has been no significant previous study of the characteristics of multi-location repairs in terms of their implications for automatic repairability. Something like: Instead, we chip away at pieces like the blind man at the elephant, or some some dumb analogy. But punchy.

To fill in this gap, and lay groundwork for more informed, systematic research efforts in the space of multi-edit repair, we conduct a systematic study of real-world bugs with multi-location patches. We look specifically at their characteristics with respect to the problem of the automatic repair search space. We study bugs curated in two real-world datasets that

[1] https://engineering.fb.com/developer-tools/getafix-how-facebook-tools-learn-to-fix-bugs-automatically/

support program repair research: Defects4J [**?**] and BEARS [**?**], in total, 1018 bugs in 51 projects. More than half of these bugs were repaired by a human developer using edits at multiple locations. We look at characteristics along each of the relevant axes of the program repair search problem: fault locations, mutation operators, fix code, and evaluation (or fitness or objective) using test cases, and find both important and, in some cases, unintuitive, implications for automated techniques. Our findings and contributions are:

- **Fault Localization.** We find that 58% of multi-edit bugs have faulty locations that are not covered by all failing tests. This is important because the assumptions underlying spectrum-based fault localization may not hold when used off-the-shelf for multi-edit bug repair, motivating the development of novel localization techniques.
- **Edit dependency.** We find that 45% of bug patches contain dependent edits, and that such bugs may be more difficult for current, state-of-the-art techniques to automatically repair. This means that APR techniques assuming edit independence [**?**], [**?**] are applicable to a nontrivial propertion of complex bugs, while still motivating development of new analyses that can reason about multiple, related edits at once.
- **Cloning in multi-location repair** We find that over 30% of bugs with multi-location patches have very similar edits applied to different locations, suggesting the viability of program repair techniques designed to utilize code clones. Moreover, importantly, bugs in which none of the faulty locations are covered by all failing tests tend to include such clones in their patches. This suggests a path forward for predicting, a priori, which type of technique may apply to a given bug.
- **Test cases for multi-edit patch validation.** We find that over 40% of bugs with multi-location patches in Defects4J and over half in BEARS do not require the edits at every location in their provided human patches to pass all test cases, suggesting that either patches contain unnecessary edits or that test cases do not fully capture the desired behavioral specification. Think about how to ephrase this a bit more intuitively, and consider breaking up this bullet into multiple findings. We also find that test case based validation methods can positively identify close to 40% of partial repairs, while less than 20% of partial repairs cause more test assertions to fail. Additionally, the *granularity* at which test suite behavior is measured (i.e., at the class, method, or assertion level) influences ability to identify partial repairs.
- dataset replication etc FIXME: Our dataset and analysis results, linked here anonymously, will be prettier for the camera ready we promise.

In Section II, we overview the problem domain to motivate/outline our research questions, and characterize our datasets. We then describe the results of our study along the three key axes of program repair: fault localization (Section IV), mutation operators and fix code (Section V), and test cases as the patch validation mechanism (Section VI). We conclude by outlining limitations (Section VII), related work (Section VIII), and summarizing discussion (Section X).

II. PROBLEM DEFINITION

In this section we overview the background of automated program repair (APR) and introduce key concepts that directly motivate our research questions.

A. Automated Program Repair: a Search Problem

The central goal of source-level automated program repair (APR) is to automatically generate patches for bugs in programs. We restrict attention to *dynamic* or *test-case guided* program repair, a prevalent class of research techniques over the past decade [**?**]. Dynamic APR techniques are characterized by their use of test cases that serve as the oracle for program correctness, at least one of which is failing (i.e., exercises the bug to be repaired).

At a high level, source level program repair is a *search problem*, where the objective is a set of source-level program edits that will cause it to pass all of the provided tests. To do this, they first *generate* a candidate patch, then run the test suite to *validate* it; for this reason they are often called *generate and validate* repair techniques. Generating a patch requires navigating a search space, which is typically defined along the following axes [**?**], [**?**]:

1) *Fault space.* The first problem with generating a patch concerns *where* in the code a patch should be applied. Most dynamic repair techniques begin by using test cases as input to a fault localization technique. Such techniques identify (and typically score) suspicious code based on which test cases execute which pieces of code. Although the particulars of the fault localization employed can vary, most APR techniques use some variant of spectrum-based fault localization (SBFL) [**?**] in this stage. The resulting computation defines the *fault space*, or the candidate code locations considered for repair.

2) *Mutation space.* After identifying a faulty program location, an APR technique must choose from the set of applicable modifications at the given point. Examples include GenProg's *append*, *replace*, and *delete* mutation operators over statements [**?**]; PAR's human patch inspired edit templates [**?**]; and Nopol's condition replacement over `if` statements [**?**]. Although a larger mutation space allows the generation of a wider variety of patches with potentially more repairs, adding more mutations grows the entire generation search space combinatorially [**?**].

3) *Fix space.* Certain mutations must be instantiated with generated or selected code before they can be applied. For example, if a technique elects to insert a null pointer check, the specific object being checked for `null` must be chosen to generate a patch. This search space that must be navigated to instantiate a patch is referred to as the *fix space*. The fix space can be quite large (or even infinite), and different APR techniques use different strategies to make it tractable. Some techniques tackle the problem by taking advantage of the *plastic surgery hypothesis* [**?**], which assumes that a bug can be

repaired by code available in other parts of the same program. By contrast, learning-based approaches use models of code or repairs to inform modifications [**?**], whereas synthesis-based approaches constrain a synthesis engine to a small vocabulary of fix ingredients, possibly informed by the code near the selected faulty location [**?**], [**?**].

After generation, patches must be validated. Candidate patches are typically validated using the provided test cases as the objective function. In techniques that require one, test cases can also be used as the *fitness function* [**?**]; although some have supplemented the test case objective with partial correctness measures like program invariants [**?**], memory snapshots [**?**], or similarity to human edits [**?**], with maximization of tests passed maintained as one of possibly many objectives.

B. Multi-Location Patches and Dataset

note to self/Claire: make sure this is in here somewhere, it's good Over 50% of the fixes in four Apache projects involve two or more entities – i.e., a Java class, method, or field – and 66%-76% of those multi-entity fixes involved syntactic dependencies [**?**].

State-of-the-art dynamic APR techniques typically generate small patches that are confined to a small portion of a program's code, rather than larger patches that span multiple logical units of code (e.g., multiple control flow blocks, functions, or files). The primary motivation of this study is to understand what makes the generation of larger patches difficult, and identify properties of larger human-written patches that can be exploited by future APR tools.

More precisely, this work focuses on *multi-location* patches. There are several plausible definitions of multi- vs. single-location patches, with implications for how they are studied. For the purpose of this study, we define a *patch location* as a contiguous sequence of edited lines of code. We combine two locations if one simply opens or closes a syntactic block inserted by the other. That is, conceptually:

```
1 + Some_edit {
2 + New code
3   Existing code
4 +}
```

is treated as a single-location edit in our study. We ignore changes to comments, whitespace, or `import` statements.

We believe this is an intuitive definition consistent with the general APR paradigm. As a shorthand, we refer to bugs with multi-location human-written patches as *multi-location bugs*.

Decomposing the problem of dynamic APR into multiple subproblems allows us to ask research questions specific to each. First, we investigate the prevalence of human-written multi-location patches in Section III. Then we investigate properties of the multi-location fault space (Section IV), mutation and fix spaces (Section V), and validation (Section VI). We discuss the implications of our findings on multi-location repair in Section IX.

III. Dataset Characteristics

Our study requires a dataset of indicative, real-world, multi-location defects. We study both the defects in Defects4J v2.0.0 [**?**] and BEARS [**?**]. Table I summarizes these datasets, both of which consist of historical bugs found in real world software projects. Defects4J contains 835 bugs from six Java software projects, and is a popular dataset for evaluating program repair tools that target Java [**?**]. The dataset's patches are manually minimized to isolate the bug fix and exclude non-repair edits such as refactorings and feature additions.

With any dataset, however, there is a risk associated that tools may overfit to the defects in question, and there is evidence that this situation applies to program repair and Defects4J [**?**]. We thus also study bugs from BEARS [**?**], a set of Java bugs derived from failed Travis-CI builds of GitHub projects. BEARS offers 251 bugs from 72 software projects, providing a greater diversity of projects compared to Defects4J. Several projects in BEARS, however, are structured as multi-module projects, which are not currently compatible with our automation tools. We thus limit our analysis of BEARS to 183 bugs from 34 single-module projects.

We start our analysis of multi-location patches by asking the following research question:

> **RQ1:** How prevalent are human-written multi-location patches?

Table I lists the numbers and percentages of multi-location patches in Defects4J and BEARS. We find that multi-location patches comprise over half of BEARS and almost half of Defects4J. Although a multi-location human patch for a bug does not imply the non-existence of a simpler patch, the high proportion of bugs that have multi-location patches demonstrates the relevance of such bugs to fault localization and program repair.

BEARS contains a greater proportion of multi-location patches compared to Defects4J. This may be the result of manual patch minimization in Defects4J [**?**] and lack thereof in BEARS. Thus, some BEARS patches may be multi-location as a result of additional changes added for non-repair reasons.

IV. Fault Localization

Spectrum-based fault localization (SBFL) is the most commonly studied dynamic fault localization technique, and studies have shown that it is more effective than other techniques such as Mutation-based fault localization [**?**] or dynamic program slicing [**?**]. It is a key first step to characterizing the *fault space* in automatic program repair, narrowing it to a portion of the program most likely to correspond to the fault.

Fundamentally, the core assumption underlying SBFL is that *failing tests execute buggy portions of the code relatively more often than passing tests.* Thus, if all failing tests execute a particular line of code, then that line of code is highly suspicious. SBFL techniques compute a suspiciousness by measuring how often a line is executed by failing tests as

Project	Bugs	Src (kloc)	Test (kloc)	Multi-location bugs		Multi-test bugs		Multi-location & Multi-test bugs	
Defects4J									
JFreeChart	26	193.3	74.6	11	42%	10	38%	7	27%
Closure compiler	174	150.6	112.6	55	41%	58	44%	31	23%
Apache commons-lang	64	57.8	47.4	32	49%	17	26%	13	20%
Apache commons-math	106	45.0	41.5	53	50%	28	26%	22	21%
Mockito	38	23.0	28.5	16	42%	18	47%	8	21%
Joda-Time	26	82.9	70.4	17	63%	13	48%	9	33%
Apache commons-cli	39	5.7	4.6						
Apache commons-codec	18	5.8	5.8						
Apache commons-collections	4	64.9	47.6						
Apache commons-compress	47	12.8	3.1						
Apache commons-csv	16	2.6	3.9						
Gson	18	16.8	12.8						
FasterXML jackson-core	26	27.9	8.1						
FasterXML jackson-databind	112	80.7	41.2						
FasterXML jackson-dataformat-xml	6	7.5	6.2						
Jsoup	93	7.9	2.2						
Apache commons-jxpath	22	28.7	7.6						
All (Defects4J)	835	813.9	518.1	184	47%	144	36%	90	23%
BEARS (single-module)									
FasterXML jackson-databind	26	80.7	41.2	17	65%	4	15%	2	8%
INRIA Spoon	62	66.2	30.8	39	63%	23	37%	18	29%
spring-data-commons	15	45.8	28.8	9	60%	6	40%	2	13%
traccar-traccar	42	47.9	8.6	24	57%	3	7%	2	5%
30 other projects	38	—	—	22	58%	36	95%	9	24%
All (BEARS)	183	>240.6	>109.4	111	61%	72	39%	33	18%
Combined (Defects4J & BEARS)	1018	>1054.5	>627.5	295	51%	216	37%	123	21%

TABLE I: Characteristics of the Defects4J (top) and BEARS (bottom) datasets.

compared to passing tests. This suspiciousness score can be calculated a few different ways, but is typically a linear combination of the passing and failing test coverage. One of the oldest and commonly studied SBFL technique is Tarantula [**?**]. In Tarantula, the suspiciousness score for a line s is calculated by:

$$susp(s) = \frac{\%F(s)}{\%F(s) + \%P(s)}$$

where $\%F(s)$ and $\%P(s)$ are, respectively, the percentage of failing tests and passing tests that execute s. There are newer, more effective SBFL techniques that calculate this score differently, such as Ochiai [**?**] and DStar [**?**]. Both of these were included in an empirical study comparing fault localization techniques and were found to localize a similar set of faults [**?**].

Assigning a suspiciousness score to each line of code is well-suited to single location repair. Indeed, the evaluation of most SBFL techniques asks exactly the question of interest when considering a technique's suitability for single-location repair: how often does a given technique assign high scores to individual buggy lines of code?

Such evaluations, by and large, do not consider the implications of suspiciousness scoring in a multi-location repair context. Instead, evaluations typically consider a technique "successful" if it identifies *any* of a set of changed lines as highly-ranked or likely-suspicious. While appropriate for the question being asked in such evaluations, this does not address suitability for multi-location program repair. Identifying one of several buggy locations is generally inadequate in a context where multiple locations must be modified. In order to investigate how well the SBFL assumption applies to tests that identify bugs that require multi-location patches, we ask the following research question:

RQ2: Do failing tests of a multi-location bug execute all the same patch locations or all different patch locations?

We focus especially on bugs that are associated with multiple failing tests: a bug with only a single failing test trivially and equally implicates all the lines that test executes. If multiple tests cover the modified locations well, then SBFL's core assumption holds and multi-location repair can expect to effectively make use of the off-the-shelf ranking these techniques currently provide (indeed, this has been tried [**?**]). If not—that is, if multiple tests exercise *different* portions of the buggy code—SBFL off-the-shelf will by definition be less effective in guiding APR to correctly modifying multiple buggy locations at once. Fundamentally, we ask whether multiple failing tests cover exactly the same patch locations, exactly disjoint patch locations, or some combination.

In addition, for purposes of fault localization, we want to know if we can detect whether tests will cover disjoint or same

fault locations given only the buggy code. Thus, we also ask:

> **RQ3:** Do tests that cover different (resp. same) fault locations also cover different (resp. same) code locations in general?

A. Methodology

Between both datasets, there are 191 total bugs that both require multi-location patches and contain multiple failing tests. However, we were not able to obtain coverage data for one of the bugs in Bears, leaving 190 total bugs: 158 in Defects4J, and 32 in Bears. For each of these bugs, we used JaCoCo[2] to determine which code locations in both the buggy and patched versions were executed (at least once) by each failing test.

To answer RQ2, we used the patch locations that were executed by the tests to categorize each bug into three *coverage patterns*, as follows:

- *disjoint* bugs are those for which no line in the patch is covered by all failing tests. Intuitively, these are the bugs for which the core SBFL assumption is violated.
- *overlap* bugs are those for which some patch lines are covered by all failing tests, but some are only covered by a subset of the failing tests. These bugs also violate the core assumption of SBFL, albeit to a lesser extent.
- *identical* bugs are those for which all tests cover the exact same set of patch lines.

In our experiments, we classify bugs using coverage of the patch lines, as opposed to coverage of patch locations (where a patch location is considered covered if any line at that patch location is executed). For the purposes of fault localization, coverage of patch locations is more valuable than coverage of patch lines, and we analyzed 66 Defects4J bugs[3] and all the Bears bugs, a total of 98 bugs, to see whether categorization based on patch line coverage matched categorization based on patch location coverage.

To answer RQ3, we took all code locations executed by tests in the buggy version and calculated the percentage of lines that were executed by all failing tests. A lower percentage indicates that the failing tests execute different portions of the buggy code, whereas a higher percentage indicates that the failing tests all execute a similar set of code, corresponding to bugs we expect to be *disjoint* and *identical*, respectively.

After calculating the percentages, we split the bugs based on whether we categorized the bug as *disjoint*, *identical*, or *overlap* in the previous experiment. We qualitatively observed the degree to which the percentage corresponded to its coverage pattern. Use a real statistical measure

[2]https://www.eclemma.org/jacoco/

[3]All the bugs in Defects4J version 1; we did not have enough time to do the analysis on bugs added in Defects4J version 2.

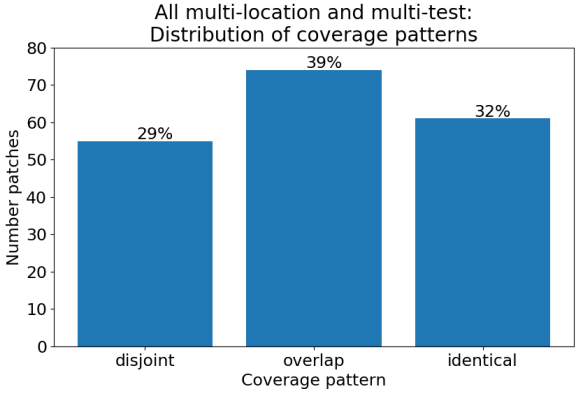
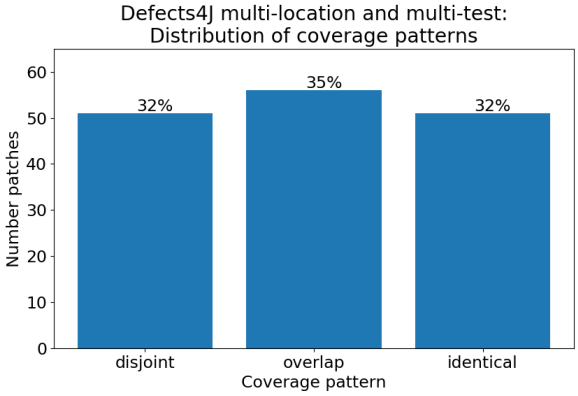
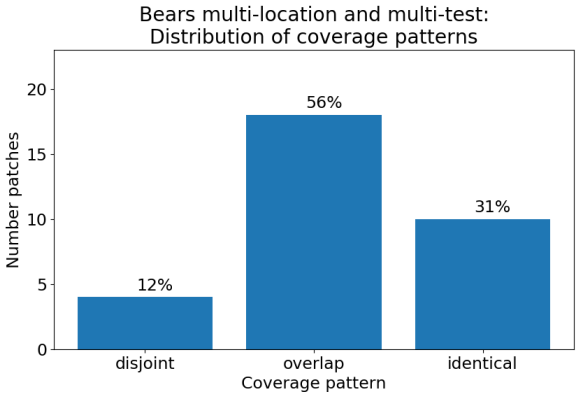


Fig. 1: Distribution of coverage patterns for bugs with multiple failing tests that are repaired with multi-location patches in Bears and Defects4J.



(a) Distribution of coverage patterns for Defects4J.



(b) Distribution of coverage patterns for Bears.

Fig. 2: Distribution of coverage patterns by dataset, indicating significant differences between the multi-location bugs in Bears and Defects4J.

B. Results

1) Distribution of Coverage Patterns: Figure 1 shows results for the overall distribution of coverage patterns, answering RQ2. For a 29% bugs, none of the patch lines were executed by all failing tests as they were shown to be *disjoint*. An almost equal proportion of bugs, 32%, were classified as *identical*, indicating that all failing tests executed the same patch lines. In addition, another 39% were classified as *overlap*, exhibiting a combination of the *disjoint* and *identical* characteristics.

Note, however, that the behavior varies considerably by dataset; Figure 2 shows results. In Defects4J, the patterns all have similar numbers of bugs, while in Bears, there are fewer *disjoint* bugs and more *overlap* bugs. We hypothesize that this may be due to differences in how the two datasets were selected and constructed: The Defects4J dataset specifically enforces a requirement that the patches in the dataset be isolated, i.e., not containing any refactorings or new features, to improve the usability of the dataset. The authors specifically chose patches that met this requirement, and in some cases, manually isolated the bug themselves [**?**]. By contrast, the bugs in Bears are scraped directly from continuous integration systems, and the only requirements for inclusion is that the bug must be reproducible and that the patch must be written by a human. In addition, Bears was designed to be evolvable and relatively easily expanded as a dataset, which is at odds with manual inspection and isolation of bugs [**?**]. Given that these two datasets were designed with different values and demonstrate very different behavior, these findings highlight the importance of using diverse datasets in evaluating program repair techniques.

Out of the 98 bugs we checked, only seven bugs had differing coverage patterns when classified based on coverage of patch lines vs. coverage of patch locations. All seven of these bugs were classified as *overlap* when classified by line coverage, but were classified as *identical* when classifying by patch coverage, indicating that all the failing tests were executing different paths within the same patch locations. Is this discussion confusingly worded? Because patch locations refer to the patch chunk, made up of one or more lines of code.

Overall, SBFL assumes that faulty locations are executed more often by identifying or failing test cases and is not designed to find many of these multi-location faults. Our results suggest that off-the-shelf SBFL techniques are not well-suited to guiding APR techniques that conform to the dominant paradigm to repairing these types of multi-location, multi-test bugs.

2) Generalization of Coverage Patterns: Our results in Figure 3 answer RQ3. Here, we see three distributions, separated by coverage pattern, and plotted based on the percentage of lines executed by all failing tests.

The distributions of *disjoint* bugs (and to a lesser extent, *overlap* bugs) are distributed almost uniformly across the 0% to 100%, indicating that *disjoint* bugs are not more or less likely to have failing tests that cover the same or different parts of the code base. Note that a bug categorized as *disjoint* can be scored 100%, as the patch can introduce if statements that change the control flow.

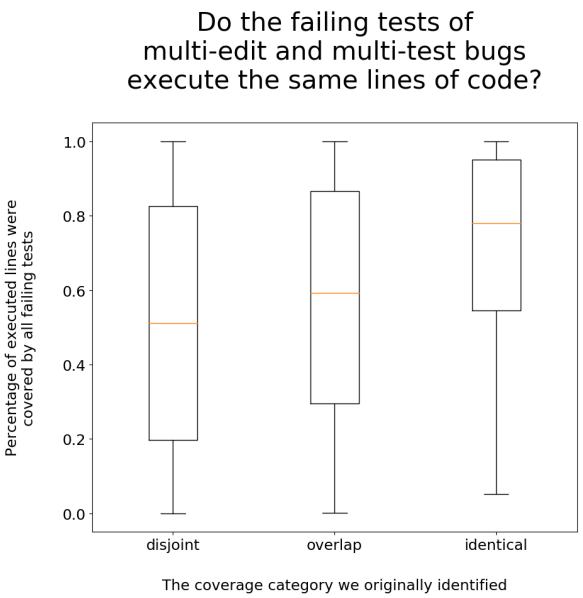


Fig. 3: Boxplots representing the distribution of bugs based on it's percentage of lines executed by all failing tests. A bug scored at 100% indicates that all failing tests executed the same lines of code, whereas a bug scored at 0% indicates failing tests all executed different lines of code. These boxplots are split by coverage pattern, as categorized before.

In contrast, we can qualitatively observe that *identical* bugs are more likely to have failing tests that execute the same lines of code, as we might expect. However, this is not a definitive characteristic, as *identical* bugs in our dataset scored anywhere between 5% to 100%.

From these observations, we conclude that the degree to which failing tests cover the buggy code are not effective predictors for the eventual coverage pattern for the bug.

V. Mutation Operators and Fix Code

Given suitably selected fault locations, APR techniques vary in the types of mutation operators they consider, how they select between them, and how they select new fix code to instantiate them, as necessary. For example, a naïve approach with only `insert`, `replace`, and `delete` operators must choose between them at a location and, in the case of `insert` and `replace`, choose code to insert/replace at that location. The few techniques that handle or at least enable multi-edit patches vary in their handling of mutation operator selection and instantiation. At one extreme, semantics-based repair [**?**], [**?**] can represent dependent edits between multiple locations as a conjunction of multiple constraints to simultaneously solve, bounded by some number of edits that are computationally feasible, while restricted to a relatively small library of possible code components for use in the inductive synthesis

problem. At the other extreme, search-based or evolutionary techniques [**?**], [**?**] typically treat different mutation operators independently. That is, a modification in one location does not inform the selection of a modifications to apply in a second location; instead, the heuristic search is trusted to identify copacetic combinations. The size of the search space increases combinatorially in this context, however, rendering the chances of finding suitable multi-edit repairs without additional guidance quite low [**?**], [**?**]. Accordingly, heuristic techniques targeted at multi-edit repair contexts [**?**] make assumptions about the shape of the search space to render it tractably constrained — in particular, targeting bugs that can be repaired by multiple syntactically similar pieces of fix code.

These kinds of targeted techniques surface general questions about the *relationship* between multiple edits, with implications for how edits should be designed for automatic multi-location repair.

A. Dependencies

One of the dimensions of the relationship between multi-location patches is potential *dependency* between the edits. Specifically, we ask the following research question:

> **RQ3:** How prevalent are dependencies between edited code?

To answer this question, we broaden the scope of multi-location patches to include all patches containing at least two added, removed, or modified lines, ignoring edits to comments, whitespace, or imports. This expanded dataset contains 659 Defects4J v2 and 151 BEARS bugs. We consider a patch to contain dependent edits if there exists control or data dependencies between added, deleted, or modified statements. We analyze deleted/modified statements in the pre-patch code and added/modified statements in the post-patch code for dependencies.

For practical performance and scalability reasons, we perform intraprocedural analysis. We do, however, gather some interprocedural data dependency information using the following heuristics:

- If a statement invokes a method, then we assume that the statement reads all variables used in the method arguments.
- If a statement invokes a getter method `Class.getX()` (for any `Class` and `X`), then we heuristically assume that the statement reads `Class.X`. Note that `Class.X` does not need to actually exist.
- If a statement invokes a setter method `Class.setX()`, then we heuristically assume that the statement writes to `Class.X`.

a) Results: We find that 40% of Defects4J patches, 63% of BEARS patches, and 45% of the combined datasets' patches contain dependencies between edited statements. Our finding supports earlier results on edit dependencies in bug patches [**?**].

Patches with Dependent Edits						
APR (RepairThemAll)	Defects4J		BEARS		Combined	
Success by any tool	43	32%	9	9%	52	23%
Failure by all tools	93	68%	86	91%	179	77%
Not evaluated	130	—	0	—	130	—
Total evaluated	136	100%	95	100%	231	100%
Total	266	—	95	—	361	—
Patches without Dependent Edits						
Success by any tool	96	57%	7	13%	103	46%
Failure by all tools	73	43%	49	87%	122	54%
Not evaluated	224	—	0	—	224	—
Total evaluated	169	100%	56	100%	225	100%
Total	393	—	56	—	449	—

TABLE II: Multi-line patches with respect to the presence of dependent edits and whether any APR tool successfully repaired the bug in RepairThemAll [**?**]. Bugs categorized as Not evaluated are those introduced in v2 of Defects4J, which RepairThemAll didn't study.

Patches with and without edit dependencies both form a substantial portion of multi-line patches, and neither type of patches should be ignored.

We further compare how often APR techniques successfully repair bugs whose patches do (not) contain edit dependencies. Table II show the frequencies and percentages of multi-line patches with respect to edit dependency and whether an APR tool successfully repaired bug in RepairThemAll [**?**], a repair experiment running 11 APR tools on 5 benchmarks, including Defects4J v1 (containing a subset of Defects4J v2) and BEARS.

We find that the presence of edit dependencies reduces the likelihood of APR tool success for Defects4J bugs. Using a χ^2 test, we find a statistically significant relationship ($p < 0.001$) between APR success and the presence of edit dependencies for Defects4J patches. We fail to find statistically significant relationships over BEARS patches, likely due to the small number (16) of successfully auto-repaired BEARS bugs with multi-line human patches. The generally lower auto-repairability of bugs with edit dependent patches compared to their non-edit dependent brethren suggest that such dependencies indeed add complexity to the search for a repair.

Our results substantiate prior research [**?**] on the frequency of bugs with dependent edits in human patches. A diverse range of APR techniques are less likely to repair such bugs. Dependent edits, however, may be an opportunity to constrain the search space by creating constraints between otherwise independent mutations. There is an opportunity to profit from edit dependencies to repair a large class of difficult bugs.

B. Cloned code

In addition to dependency, multi-location patches can be related through syntactic similarity. That is, a patch can consist of a code edit applied repeatedly in multiple locations. Thus, we ask the following research question:

RQ4: How often do code clones occur in multi-location bugs? Is the existence of code clones in human patches correlated with specific patterns of fault localization?

Previous work [**?**] suggests that one potential fruitful way to enhance APR techniques is to allow them to apply a single edit to multiple locations. This is based on the observation that human developers tend to make exactly the same edits in multiple locations when fixing bugs.

As a proxy for measuring how often this style of repair actually occurs, we seek to understand the prevalence of code clones in human patches. If a large proportion of multi-location bugs contains code clones, then we can corroborate the usefulness of previous repair techniques that specifically apply similar edits for a bug [**?**].

We further attempt to correlate the coverage patterns outlines in Section IV with the existence of code clones in human patches. Our intuition is that a *disjoint* bug may be more likely to occur when a developer needs to apply the same fix at multiple independent locations, that can or should be tested separately. We therefore hypothesize that *disjoint* bugs will have a higher incidence of code clones. By contrast, bugs categorized as *same* or *overlap* may have more interrelated parts that are not merely the same statements copied to multiple locations.

1) Methodology: Our dataset includes all multi-location bugs excluding those with more than 6 faulty locations to constrain the search space, leaving 216 bugs to analyze. We determine the existence of code clones through manual inspection.

In comparing these results with the results of the coverage experiment in Section IV, we focus on the multi-test and multi-location bugs used in the coverage experiment and exclude bugs with more than 6 faulty locations, leaving 166 bugs.

We call the edits at two locations code clones if it is one of the following four cases:

1) Same Case: The two locations are alpha-equivalent.
2) Literal Case: The two locations differ by at most one constant or arithmetic operator, or replacement of one constant with variable.
3) Composite Case: The patch at one location is exactly copied and contained within the patch at the second location (the second location has additional lines in the patch).
4) Move Case: Two locations forms a "movement" of code (i.e., one location is an insertion of code while the other is a deletion of the same code, essentially moving the code from one location to another).

2) Results: As shown in Table III, out of the 372 multi-location bugs, 121 of them, or 32.5%, included at least one type of cloning between fault locations, indicating a significant prevalence of code clones in multi-location human patches. Note that "Any" may not be the sum of the previous four rows because some bug could contain multiple pairs of code clones that belong to different categories.

	Defects4J	BEARS	Combined
Same	71	9	80
Literal	23	2	25
Composite	11	1	12
Move	11	1	12
Any	108	13	121
No clones	200	51	251
Total	308	64	372
% with Clones	35.1%	20.3%	32.5%

TABLE III: Code clone information for multi-location bugs. *Same*, *Literal*, *Composite* and *Move* correspond to the four types of code clones considered, as described in Section V-B1. "Any" means the bug contains any of the four types of clone; some bugs contain pairs of clones that belong to different categories. Over 30% of multi-location bugs has code clones.

	Same Method	Same Class	Diff Class	Total
Same	35	33	12	80
Literal	12	8	5	25
Composite	4	8	0	12
Move	9	1	2	12
Total	60	50	19	129

TABLE IV: Relative locations of sets of code clones in each category (defined in V-B1). Same Method denotes that the code clones occur in the same method, Same Class denotes that the code clones are in the same class but not in the same method, and Diff Class denotes that the code clones are in different classes. The majority of code clones occur within the same method, and are of the "Same" category.

As shown in Table IV, close to half of code clones are in the same method, while less than 15% of the code clones occur in different classes. Moreover, out of the 129 sets of code clones, 80 of them belongs to the "same" category, which means that 62% of sets of code clones have completely alpha-equivalent edit locations within them.

In Table V, we show the incidence of code clones among the three coverage patterns identified in Section IV. Note that out of 372 bugs evaluated for code clones, only 166 of them have multiple failing tests, so only these bugs are included in the table. Out of 48 bugs labeled as *disjoint* in the coverage, over half of them has code clones. In contrast, the *overlap* and *identical* categories respectively had 20% and 12% bugs with

	Disjoint	Overlap	Identical	Total
Has Clone	25	12	7	44
No Clone	23	48	51	122
Total	48	60	58	166

TABLE V: Multi-location and multi-test bugs categorized by coverage pattern and presence of clones; definitions for Overlap, Disjoint, and Identical are taken from our experiments on fault localization.

code clones, This indicates that a bug with *disjoint* coverage result is more likely to contain code clones in its human patch than bugs with other coverage results.

This result qualitatively supports the hypothesis that *disjoint* bugs are more likely to contain code clones, while the other two classifications have comparatively less code clones. This has potential implications for technique design: that is, if there appears to be relatively less coverage overlap between multiple tests, it is possible that, if multi-location, the bug may be *disjoint*. If so, applying the same edits at multiple locations may be more likely to succeed. We leave a concrete investigation of this possibility to future work.

VI. Test case-based validation

Dynamic program repair uses test cases to validate patch plausibility, and tests are often used as a proxy for a full correctness specification. Program repair techniques can suffer from *overfitting* — that is, they can produce patches that satisfy the provided tests, but do not generalize to the higher-level specification [**?**]. Indeed, not all edits in a human patch are always necessary to pass all tests. This could be due to a human patch including refactoring or other changes that does not actually change code behavior [**?**], [**?**], or it could be a sign that the test suite is inadequate. Thus, the adequacy of the test cases and their suitability for identifying patch correctness is a key problem in program repair, and we study it specifically with respect to multi-location repair by asking:

RQ5: How often are multi-location bugs minimal with respect to their provided test cases?

In evolutionary or search-based approaches, test case quality is even more pressing, as they are typically used to define an objective or fitness function. In these contexts, ideally, test cases could usefully identify partial solutions [**?**], or subsets of edits that comprise the eventual patch. Whether test cases are suitable for this purpose in a multi-edit context is a matter of debate that motivates some techniques to simply target single-edit bugs [**?**], [**?**]. We study this phenomenon directly for multi-location bugs:

RQ6: How well do test case based validation methods identify partial repairs?

Given a buggy program and a multi-location repair, if we apply a part of the valid repair to the buggy program (a partial repair), then semantically, the partial repair should be considered closer to the full repair compared to the original. Ideally, test cases would suitably identify partial repairs as "closer" to a full repair than the original (unmodified program). This could provide suggestions for composing partial repairs into full repairs (or guidance for an evolutionary search algorithm attempting the same). However, sometimes partial repairs will perform no differently on unit tests compared to the original program [**?**], [**?**], and other times they could perform worse [**?**]. We want to find out how often each of these situations happen.

Moreover, unit test performance can be measured in different granularity levels. We would like to compare the performance of the fitness functions at identifying partial repairs at different granularity levels.

A. Methodology

For each bug in Defects4J (excluding Closure, which uses a non-standard testing setup) and BEARS (single-module only), we generate and test all combinations of partial repairs, as follows:

1) Partial Repairs: To construct partial repairs for each bug, we apply a non-empty subset of location-level edits from the human patch to the broken code. In order to minimize the number of syntactically invalid partial repairs, we pre-add, prevent the deletion of, or prevent the narrowing of scope of import statements, helper methods, helper classes, and variable declarations. For example, if a patch contains one edit location that declares a new variable `var` and a second edit location that might use `var`, then we pre-add the declaration of `var` from the first edit location. This allows us to examine semantic behavior of partial edits directly.

We do not split edit locations into two when applying any of our preprocessing steps. If an edit location, however, only contains edits that we pre-apply or remove during preprocessing, then we discard the now-empty edit location from the set of potential edits to apply. We eliminate bugs whose human patches contain only one edit location after preprocessing (for example, a 2-location patch that adds a new helper method in one edit location and invokes the helper method in the other).

Due to the exponential growth of the number of partial repairs with respect to the number of available edit operations, we evaluate bugs whose patches contain between two and six edit locations after preprocessing. Moreover, we exclude Defect4J's Closure bugs due to their nonstandard test suite setup. We generate and evaluate 1884 Defects4J and 596 BEARS partial repairs, derived from 240 Defects4J and 74 BEARS bugs.

2) Test Granularity: There are different ways to evaluate and compare JUnit test results. Class level granularity looks at which test classes passed and which failed; at method-level granularity, we look at which test methods passed/failed. We also consider *assertion-level* granularity, defined as follows: for each test method M, let $A(M)$ be the set of all assert statements in M. When M is run, if an assertion failed, the failure is recorded and the method is allowed to continue to run (normally, the test method throws an error and terminates). After running the method, for each assert statement $a \in A(M)$, let $b(a)$ be 1 if a never failed once during the running of M, and 0 otherwise. We define the assertion score of M to be $AS(M) = \frac{\Sigma_{a \in A(M)} b(a)}{|A(M)|}$. If M failed to run to completion due to timeouts or exceptions that are not related to assertions, then we define $AS(M) = 0$. Thus by definition, $AS(M) = 1$ if M passes. If a program passed more assertions in M, there should be an increase in $AS(M)$.

		Defects4J		BEARS	
Minimized?		No	Yes	No	Yes
Positive	Class	25.95%	12.77 %	27.60%	9.22%
	Method	40.82%	34.58 %	28.41 %	18.23%
	Assertion	44.49%	39.93 %	28.68 %	18.69%
Neutral	Class	59.56%	68.30 %	58.97 %	73.31%
	Method	39.62%	40.05 %	53.08 %	64.30%
	Assertion	33.73%	31.52%	49.52 %	59.60%
Negative	Class	9.30%	11.57 %	8.78 %	13.30%
	Method	13.51%	17.16 %	10.13 %	13.30%
	Assertion	15.37%	19.80 %	12.75 %	16.41%

TABLE VI: Weighed percent of partial repairs exhibiting a Positive, Neutral, or Negative change in test passage, measured at different levels of granularity and with both minimized and unminimized patches.

	Disjoint	Overlap	Identical	Total
$\exists$ positive partial repair	40	30	12	82
$\nexists$ positive partial repair	1	4	16	21
$\exists$ negative partial repair	2	8	16	26
$\nexists$ negative partial repair	39	26	12	77
Total	41	34	28	113

TABLE VII: Comparison of patch coverage and partial repair test passage.

B. Results

To answer RQ5, we find that 41.3% (99 out of 240) of Defects4J bugs and 52.7% (39 out of 74) of BEARS bugs do not need all edit locations in the human patches to pass all tests. A χ^2 test reveals a statistically significant ($p < 0.001$) difference in the frequency of non-minimal patches between the two datasets. Since Defects4J maintainers manually minimized patches to exclude non-corrective changes, the substantial minority of non-test minimal patches (41.3%) is further evidence of test suites' weakness as a proxy for correctness. BEARS patches, unlike Defects4J, are not minimized. Lack of patch minimization is a likely reason for the greater percentage of non-test minimal patches (52.7%).

Due to the phenomenon above, we present the results for RQ6 in two ways: treating the provided human patch as the full repair (Unminimized), and considering only the minimum set of edit locations in the provided human patch that are necessary to pass all unit tests (Minimized). Some bugs are minimized to a single edit location, and are thus excluded from the Minimized results. The minimized results include 942 Defects4J and 232 BEARS partial repairs, derived from 165 Defects4J and 44 BEARS bugs.

Since each bug may have a different number of partial repairs (based on the number of implicated locations per patch), we analyzed the results with the partial repairs *weighted* such that each bug has equal weight, i.e., for a bug with n edits, each edit has weight $\frac{1}{n}$. We then identify how many partial repairs pass more tests than the original buggy code (*positive*), the same number of tests as the original buggy code (*neutral*) or fewer tests than the buggy code (*negative*). We do this for tests at each level of granularity. Partial repairs that do not compile are excluded from the count.

Table VI shows weighted percentages of partial repairs exhibiting different changes in test suite behavior. Assertion level granularity correctively positively identifies 44.49% (unminimized) and 39.93% (minimized) of the partial repairs in Defects4J and 28.68% (unminimized) and 18.69 (minimized) of the partial repairs in BEARS. Only 33.73% (unminimized) and 31.52% (minimized) of Defects4J partial repairs and 49.52% (unminimized) and 59.60% (minimized) of BEARS partial repairs are neutral. Less than 20% of bugs in both datasets (minimized or unminimized) are negatively identified. Although the rate of positive and correct identification of partial repairs varies between datasets, tests are not often adversarial towards partial repairs, contrary to prior work [**?**].

Finer granularity levels are better at identifying partial repairs positively, but they also increase the chance of erroneously mis-identifying partial repairs. Improvements between granularity levels vary across datasets — the improvement in positively identified partial repairs between method and assertion level granularities is negligible in Bears (¡0.5%) but greater in Defects4J (3.67% unminimized and 5.35% minimized). Moreover, evaluating patches at finer granularity levels often adds overhead. Method level granularity may require executing set-up and tear-down processes for each method rather than for an entire test class. Assertion level granularity requires instrumenting JUnit with additional logic to continue execution when assertions fail. Such overhead lowers the number of candidate patches that a G&V tool can validate per unit time. The choice of a granularity level is a balancing act with implications on multiple axes of APR performance.

Table VII shows the number of bugs with 2+ negative tests and 2–6 edit locations in the minimized patch, grouped by patch coverage under fault-revealing tests and the existence of positively/negatively test-identified partial repairs at any granularity level. The vast majority of analyzed multi-test, multi-location patches have at least one positively identified partial repair. Moreover, a χ^2 test finds statistically significant relationships ($p < 0.001$) between (non-)identical patch coverage and the existence of positive and negative partial repairs. Bugs with identical coverage are less likely to have positive partial repairs and more likely to have negative partial repairs. Thus, identical coverage correlates with an *adversarial* test-based fitness landscape. Partial patches with identical coverage may be difficult for a test-based fitness function to identify.

These connections between coverage and partial repair test passage suggest that fault-revealing tests' patch coverage reflect the tests' ability to detect components of a patch. Intuitively, if a patch can be assembled from two components **A** and **B**, two tests that both fully cover **A** and **B** are less effective at assessing partial correctness than two tests that only cover **A** and **B** separately. In the case of identical coverage, the presence (or absence) of either **A** or **B** interferes

with both tests; in the case of disjoint coverage, the presence of **A** or **B** affects only its covering test. Coverage thus influences the accuracy of tests as a judge of partial correctness.

VII. Limitations

Coverage. We use JaCoCo to collect coverage for experiments, but it has some limitations. For example, it cannot collect coverage information for inserted cases for `switch` statements, `else` statements, method signatures, and other code constructs that are compiled away during the transformation to bytecode. Since our focus is on how the coverage of different failing tests compared to each other, we felt that JaCoCo's coverage was a good enough approximation for these experiments.

Dependency analysis. We analyzed dependencies intraprocedurally with some interprocedural heuristics, which make assumptions derived from common Java practices. These assumptions are not always true. We use heuristics to add some approximated interprocedural data while retaining interprocedural analysis's scalability to large, real world software.

Code clones. We only examined code clones at the location level. However, code clones can also occur at the granularity of individual lines [**?**], or even subexpressions [**?**]. We also classified clones into four categories (Same, Literal, Composite, Move), but it is possible to decompose them in other ways. Additionally we did not distinguish between the number of edit locations in a particular set of clones. It is possible that a set of clones could contain two or more locations with similar edits, which we do not examine.

Partial repair construction. Constructing partial repairs, as described in Section VI-A, breaks up multi-location patches into single-location edits. In reality, most APR techniques mutate code at a finer granularity, so these edits may not be representative. Additionally, we only observe if fitness functions can identify partial repairs. Perhaps the partial repairs in our experiments are outside of the search space of a particular APR technique. Regardless of this limitation, this experiment provides valuable insight into the challenge of automatically constructing multi-location patches.

VIII. Related Work

Recent empirical studies on fault localization find that SBFL is a more effective fault localization technique than a host of other techniques, such as mutation-based fault localization [**?**], [**?**], program slicing, predicate switching, information retrieval, and other techniques. However, a fault localization technique that outperforms all of these can be created by using machine learning to combine multiple fault localization techniques, implying that different fault localization techniques can be used to localize different kinds of faults [**?**]. Most empirical studies on fault localization evaluate techniques by determining whether the technique can localize any faulty line. This usefully evaluates single line or single location faults, but not necessarily multi-location faults.

Mutation testing contains parallels to program repair, as seen in G&V program repair [**?**] and fault localization techniques [**?**], [**?**], [**?**]. With regards to multi-location bugs, there has been recent work on higher order mutants, which is interested in finding certain combinations of mutants with particular behaviors. Higher order mutation testing shares similar difficulties with multi-location bugs, particularly due to exponential growth of an already large search space [**?**]. The current state of the art in finding specific classes of higher order mutants is using search based approaches; in particular, the best approach currently identified is a genetic search, which is reminiscent of program repair [**?**], [**?**].

Qi et al. [**?**] evaluated the patches generated by three generate-and-validate repair tools [**?**], [**?**], [**?**]. They found the vast majority of generated patches to be incorrect and equivalent to a single functionality deletion. Patch *overfitting* to the provided test cases can be measured via the use of held-out test suites [**?**], a technique that empirically validates the challenges of producing high-quality repairs in response to a single test suite. Later work found patch incorrectness to be also problematic in Defects4J [**?**] and in semantics-based repair techniques [**?**]. We similarly find that test suites provide imperfect semantic coverage over multi-location edits: many bugs can be fixed with only a subset of the edits included in a multi-location human patch. We also find, however, that test cases can often identify partial patches.

Long and Rinard [**?**] studied the prevalence of correct and incorrect plausible patches in the search spaces of SPR [**?**] and Prophet [**?**]. They found incorrect plausible patches to outnumber correct patches by orders of magnitude. When they increased the search space by adding additional mutation operations, they found an increased number of correct patches, but APR tool performance might actually degrade due to a simultaneous increase in incorrect plausible patches and the combinatorial growth of the search space. Although we do not study this problem in particular, we do similarly seek to understand the challenges imposed by the significantly larger search space that multi-location repair presents over single-location repair.

A previous empirical study on real bug fixes [**?**] studied fault localization dificulty, bug fix complexity, necessary mutations, relevance of API knowledge to those mutations, and buggy file types on more than 9000 real-world bugs collected via BUGSTAT. Both our paper and the previous study aim to provide useful guidance and insights for improving state-of-the-art APR techniques through empirical studies of bugs and bug fixes. Our study differs in that it focuses on one specific category of bugs: source file bugs that requires multiple edit actions in multiple locations to successfully repair, drawing insights on their behaviors in fault localization, fix mutations, and test-based fitness evaluations. Additionally unlike the previous study, we study the program repair benchmark datasets Defects4J and BEARS; we believe that the program repair community would derive greater benefit from a study of APR benchmarks.

Wang et al. [**?**] did an empirical study of multi-entity

changes in real bug fixes (where each entity is a class, method or field). Their research questions mostly focused on how often and why do real-world bug fixes have multi-entity changes, the relationship between co-changed entities, and the recurring patterns of those multi-entity changes. Through analyzing 2854 real-world bugs from four projects, they found that 66%-76% multi-entity fixes are closely related to each other via syntactic dependencies, and they identified three major recurring patterns that connects co-changed entities. They suggested a potential way to close the gap between APR fixes and real fixes by enhancing APR to incorporate multi-entity changes. In contrast, our study on bugs that requires multiple edits to fix, where the edits may be in the same entity. We define atomic changes (single edit) differently, and we studied interactions between edits (i.e. the lines that changed in the bug fix) instead of entire entities.

Previous efforts in program repair to derive more search-guiding information during candidate patch evaluation include using program invariants [**?**], [**?**], intermediate program values [**?**], and online mutation-based fault localization [**?**]. Some approaches require additional input, such as suspicious variables [**?**] or known patches for the bug under repair [**?**], while others exhibit limited performance improvements [**?**], [**?**]. Our work shows the potential benefit of using more granular objective functions such as assertion level granularity compared to class/method level granularity, and maybe future research can look into even more granular objective functions such as at a sub-assertion level (e.g: different assert distance in AssertEqual, etc).

Schulte et al. [**?**] studied software mutation robustness, in terms of how often do code mutations have no effect on test results. They found that in a large collection of off-the-shelf software, mutational robustness is about 37%, and discussed potential application of mutation robustness to proactive bug repair. Although our context is different — they studied random mutations, we study changes associated with specific bug fixes — we also find a non-trivial proportion of neutral edits in multi-location repairs.

IX. Discussion and Takeaways

Our findings demonstrate that bugs repaired with multi-location patches have fundamental differences from bugs repaired with single-location patches. These differences should be considered for designers of future APR techniques that generate multi-location bug fixes. These are our key takeaways:

- **The *core assumption* of spectrum-based fault localization does not always hold.** The core assumption of spectrum-based fault localization—that faulty locations are executed more often by negative test cases—was not true for a substantial number of bugs in our study. In our study 58% of bugs repaired with multi-location patches and characterized by multiple failing tests had patch locations that executed by some but not all tests. Additionally, 27% of these patches were disjoint (i.e., had no patch location covered by all failing tests). In these cases SBFL is unlikely to be the most appropriate choice for a fault localization technique. This finding suggests that there is still progress to be made in fault localization for automated repair.
- **Patches with dependencies are common and harder to construct.** We found that 45% of human-written multi-location patches contained intra-patch control and data dependencies. These patches are less likely to be generated by a broad range of repair techniques. This points to an opportunity to exploit dependencies by existing or new techniques to improve APR for this large class of hard-to-repair bugs.
- **Code clones are prevalent in multi-location patches.** We found code clones in 32% of bugs with multi-location patches. This observation supports recent work that leverages code clones to generate repairs [**?**]. Moreover, we found that there is a correlation between negative tests with disjoint coverage and code clones in human-written patches. This suggests the existence of a heuristic for identifying which bugs would benefit most from code clone-specific techniques.
- **Correct partial repairs rarely cause more failing tests.** Contrary to prior work [**?**], we found that test suites are infrequently hostile to partial repairs. That is, partial application of a correct patch usually does not increase the number of test case failures. This means that generate and validate APR techniques can assemble correct patches from partial repairs.
- **Patch coverage should be considered when determining the confidence of a test suite's assessment of partial correctness.** Test cases are a more accurate metric of partial correctness when coverage overlap of patch components is minimized. We can reduce the possibility of overlapping coverage by decomposing tests into smaller units. Potential methods to decompose tests include refactoring [**?**] or using a finer level of granularity for measuring test suite success, such as the assertion level granularity proposed in Section VI-A.

  This also implies that class-level granularity validation has tradeoffs. Some repair frameworks use class-level granularity for faster validation. This may come at the cost of less accurate detection of partial patches. However, this tradeoff may be worthwhile in Java, given the non-trivial challenges involved in decomposing JUnit test classes into individual methods.
- **Human patches are often not test-minimal.** Many bugs do not need all patch locations to pass all tests, offering further evidence of the incompleteness of test cases as a proxy for correctness [**?**] and the presence of non-corrective changes (e.g., refactoring, enhancements) in handwritten bug patches [**?**], [**?**].
- **Techniques need to be evaluated on diverse benchmarks.** Our two datasets, Defects4J and BEARS, exhibited different characteristics. These characteristics may explain why APR tools perform unevenly across different benchmark datasets [**?**]. Our findings provide evidence reinforcing the call for the use of diverse benchmarks when evaluating tools.

We hope that our findings and insights inspire future work to advance the state of the art of APR and take on this large, difficult class of multi-location bugs.

X. Conclusions

To date, most automated program repair techniques have been limited to bugs that can be localized to and repaired at a single program location. However, we find that 47% of bugs in the Defects4J dataset and 61% of bugs in the BEARS dataset were repaired by a human developer with multi-location patches. This motivated our study on the possibility of extending current program repair work to multi-location patches.

Our study's findings suggest deep implications for program repair targeting multi-location patches, both in terms of the applicability of existing program repair techniques and the design future repair techniques. To ensure reproducibility and encourage further investigation, we intend to release a replication package post-review.