2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

Harnessing Evolution for Multi-Hunk Program Repair

Seemanta Saha†
University of California Santa Barbara
Email: seemantasaha@cs.ucsb.edu

Ripon K. Saha, Mukul R. Prasad
Fujitsu Laboratories of America, Inc.
{rsaha, mukul}@us.fujitsu.com

***Abstract*—Despite significant advances in automatic program repair (APR) techniques over the past decade, practical deployment remains an elusive goal. One of the important challenges in this regard is the general inability of current APR techniques to produce patches that require edits in multiple locations, *i.e.,* multi-hunk patches. In this work, we present a novel APR technique that generalizes single-hunk repair techniques to include an important class of multi-hunk bugs, namely bugs that may require applying a substantially similar patch at a number of locations. We term such sets of repair locations as evolutionary siblings – similar looking code, instantiated in similar contexts, that are expected to undergo similar changes. At the heart of our proposed method is an analysis to accurately identify a set of evolutionary siblings, for a given bug. This analysis leverages three distinct sources of information, namely the test-suite spectrum, a novel code similarity analysis, and the revision history of the project. The discovered siblings are then simultaneously repaired in a similar fashion. We instantiate this technique in a tool called HERCULES and demonstrate that it is able to correctly fix 46 bugs in the Defects4J dataset, the highest of any individual APR technique to date. This includes 15 multi-hunk bugs and overall 11 bugs which have not been fixed by any other technique so far.**

***Index Terms*—automatic program repair, multi-hunk patches, code similarity**

I. INTRODUCTION

The past decade has seen significant research activity on automatic program repair (APR) techniques [1, 2]. These techniques bear the promise of helping automate the otherwise laborious process of debugging and patching bugs. However, this promise is yet to be realized in terms of practical deployment of APR techniques. One reason for this is perhaps the relatively limited classes of bugs that current state-of-the-art APR techniques can correctly fix [3, 4]. In particular, with one notable exception [5], most APR techniques are designed to target single-hunk bugs – bugs with patches confined to a single contiguous chunk of code, at a single location. However, the vast majority of bug patches span multiple hunks. For instance, 64% of the bugs in the Defects4J dataset [6] and 76% of bugs in the Bugs.jar dataset [7] require multi-hunk patches. Several previous works have acknowledged the challenges of doing multi-hunk repair [4, 5]. Any simple-minded expansion of the search space to explore general multi-hunk patches would clearly explode the repair search space.

Given a buggy program P, failing at least one test in a test suite T, an APR tool (implicitly or explicitly) searches a space S of possible mutations to P for one that allows the mutated program to pass all tests in T.

In this work, we propose an APR technique that targets a specific but important class of multi-hunk repair problems. Our solution is broadly inspired by insights from two bodies of research. The first is research on detecting and using code clones [8–17]. This research shows that code clones are plentiful in programs. Typically 5-25% [18] and as much as 50% of a subject system can be comprised of cloned code [19]. Further, replication of bugs through code clones is a common phenomenon. Up to 10% of code clones contain bugs with 55% of bugs found in code clones being replicated bugs [20]. The second body of work is APR techniques themselves, the vast majority of which, directly or indirectly, exploit the *"plastic surgery hypothesis"* – the ingredients for a repair can be obtained from existing code [21]. This can take the form of using program transformation schemas (which define the repair space) derived from a corpus of existing patches [22–27]. Alternatively, repair ingredients, such as program elements, expressions, statements, or whole snippets can be mined from existing code and re-purposed for creating a patch [27–30]. These two bodies of work point to the phenomenon that it is plausible to find "similar looking" pieces of code across a project, bearing similar kinds of bugs and warranting similar patches. Our work exploits this general insight as well.

Our proposed method generalizes single-hunk repair techniques to include bugs that may require applying a substantially similar patch at a number of locations, *i.e.,* a multi-hunk patch. We term the underlying set of repair locations as *evolutionary* [1] *siblings* – similar looking code, instantiated in similar contexts, that are expected to undergo similar changes, over the lifetime of the codebase. It is important to note, as established in RQ1 (Section VI-A) that these evolutionary siblings are not simply code clones, in the traditional sense. Further, our approach is orthogonal to use of the plastic surgery hypothesis which involves mining existing code (donor) for abstract schemas or concrete ingredients from which to compose the present repair (the donee), *i.e.,* a donor-donee relationship. By contrast, our proposed analysis seeks to find evolutionary

†This work was done when the author was an intern at Fujitsu Laboratories of America, Inc.

[1]We use evolution as a metaphor for the environment, *i.e.,* context, of a piece of code, in addition to the changes it undergoes over its lifetime.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00020

siblings exposed by the current bug, which can be repaired simultaneously and in a similar fashion. This is, in principle, independent of the actual technique used to perform the repair, and quite compatible with a donor-donee repair mechanism.

The key to our approach is to be able to accurately identify *evolutionary siblings* for a given bug. This presents the following technical challenges:

Challenge 1: Evolutionary siblings are not simply code clones. Thus, further analysis is required to expose the desired sibling relationships.

Challenge 2: The spectrum generated by the test cases may not expose or even cover all the sibling instances. Missing some of the siblings can produce a partial repair at best.

Challenge 3: Any imprecision in identifying these siblings can be potentially fatal to identifying a successful repair. Again, an under-approximation can produce a partial repair at best. Also, it is simply not computationally feasible to search the power set of an over-approximate set of potential siblings.

At the heart of our proposed method is an analysis that uses three distinct sources of information to accurately identify evolutionary siblings suitable for repair, for the bug at hand. First, it uses the test spectrum to implicate one or more of the siblings to provide a starting reference for the sibling identification. Second, it identifies all siblings of the reference sibling that have syntactic similarity but *also* semantic similarity of their context. This is done using a code similarity analysis that combines syntactic similarity with a limited scope data flow analysis to enforce similarity of the *semantic context* for identified siblings. This code similarity analysis may, in principle, identify potential siblings that are outside the scope of the test-spectra. This is an essential feature of our analysis that compensates for the weakness and incompleteness of typical test suites. Developers may often not add test-cases witnessing the bug for *each* of the siblings but rather only for some, or only one of them. Third, our method uses the revision history information to further enforce that the siblings thus identified bear a similar history of changes. This third feature discards false positives, that are not necessarily siblings in a co-evolutionary sense. Once evolutionary siblings are identified they can be handed off to any repair algorithm that should enforce *simultaneously* generating a *substantially similar repair* (modulo namespace variations) for all siblings. In principle, any traditional repair tool could be suitably retrofitted to perform this part.

We have implemented the proposed technique in a tool HERCULES [2] and evaluated it on the widely used Defects4J dataset. In our experiments HERCULES was able to correctly fix 46 bugs, the highest of any single APR technique so far. This includes 15 multi-hunk bugs, and overall 11 bugs which have not been fixed by any other technique so far.

It is noteworthy that, although not specifically discussed in [29] or [27], the implementations of ACS and SimFix are in fact capable of performing simple instances of multi-hunk repair. The bug repair counts reported in [29] and [27] include such bugs. Specifically, these tools perform multi-hunk repairs when there are separate test-cases separately implicating each bug location. Then the tool simply iteratively repairs these bugs independently, one after another. This potentially increases the search space exponentially in the number of locations, but still turns out to be viable for the simplest instances. By contrast, our approach is much more general, exploiting the sibling relationship to keep the search space effectively the same as a single-location patch. It also patches locations not covered by the test spectrum. Angelix [5] also performs multi-hunk repairs but again it does not exploit any relationship between the patched locations. The main contributions of this paper are:

- ***Technique:*** An APR technique generalizing traditional single-hunk repair to bugs that may require applying a substantially similar patch at a number of locations
- ***Analysis:*** An analysis implementing accurate detection of evolutionary siblings in service of the above repair goal
- ***Tool:*** An instantiation of the proposed repair technique in a tool HERCULES
- ***Evaluation:*** An evaluation of HERCULES on the Defects4J dataset

The rest of the paper is organized as follows. Section II presents basic background material to orient the reader, followed by a motivating example illustrating our approach in Section III. Section IV presents our proposed approach in detail. Sections V and VI present our experimental set-up and evaluation respectively. Section VII discusses the limitations of our approach. followed by a dicussion of related work in Section VIII. Section IX concludes the paper.

II. TERMINOLOGY

1) Terminology & Definition: In this paper, we consistently use the following terminology and definitions.

Single vs. multi-hunk bugs. Single-hunk bugs require program edits (insertions, deletions, or modifications) at a single location or a set of contiguous locations. Multi-hunk bugs require program edits at multiple non-contiguous locations.

Repair location. A program statement that we want to modify, delete, or insert a new statement. It should be noted a repair location may or may not be the actual buggy location.

Repair schema. An abstract program transformation template, such as adding a null checker or inserting a method invocation to a given repair location.

Candidate patch. A concrete modification to a program realized by instantiating a repair schema.

Repair space. The pool of generated candidate patches. The size of the repair space refers to the number of generated candidate patches.

Plausible patch. A plausible patch is one that simply passes all test-cases in the test suite. It should be noted that a plausible patch may still be incorrect because the test-suite may provide an incomplete specification.

Correct vs. incorrect patch. We classify a patch as *correct*, if it is semantically equivalent to the developer-provided patch,

[2]Our tool kills multi-location bugs like the mythical Hercules killed the multi-headed monster Hydra.

based on a manual examination. This is consistent with the definition used in previous work [5, 31–34]. An *incorrect patch* is a patch that is not correct.

2) Generate and Validate Repair Approach: Search-based repair approaches or so-called *generate and validate (G&V)* approaches, start with a buggy version of the program, a test-suite with at least one failing test case (revealing the bug) and one passing test case, and a set of repair schemas or program mutations to use to repair the program. A typical G&V technique operates using the following basic steps:

Step 1: Fault localization. This step produces a ranked list of repair locations. This step is typically realized using spectrum based fault localization (SBFL) techniques such as Tarantula [35], Zoltar [36], and Ochiai [37].

Step 2: Generate candidate patches. The repair approach examines each repair location in the fault localization list and applies each of the repair schemas on this statement, one at a time, in some order, to produce potential *candidate patches*. The order of repair schemas may be decided using genetic algorithms [28], random choice [38], heuristically [32, 39], or using a machine-learned model [34].

Step 3: Selection of candidate patches and validation. Each candidate patch is evaluated against the test suite, and if it passes, is output as a plausible patch. This step, which is computationally expensive, is generally optimized by first ranking candidate patches and selecting only a few of them for validation. The validation step is further optimized by first testing a candidate patch against a subset of the suite, *e.g.,* only failing tests, before executing the complete suite.

III. Motivating Example

In this section, we provide a brief overview of our approach with a motivating example, presented in Figure 1. The example is a real-world bug-fix in Apache Commons Math (Jira Official Bug ID: MATH-855), which is also a bug instance (Math-24) in the popular Defects4J dataset. The bug report says that Brent Optimizer was not always reporting the best point. As we can see in Figure 1, the assigned developer made two similar modifications at two locations to fix the bug, and the modifications are not trivial. Previously, the method was returning an object instance $current$ from two locations but actually they should be a method call $best(current, previous, isMinim)$ at each location. Now we discuss how HERCULES fixes this bug while overcoming the research challenges outlined in Section I.

The repair locations may not be part of code clones. From the developer's patch, we can easily observe that although the statements at the repair locations are the same, they are not code clones in a traditional sense. This is because the respective statements around the repair locations do not match. Generally, clone detection tools try to get a good trade-off between the number of minimum statements/tokens in a code snippet to be a clone to avoid producing a lot of false positives. Certainly, we can detect clones at a statement level. However, in that case, we would get many false positives. For our example, if we search the entire code-base using `return current`, we will get 10 instances. Furthermore, if we abstract the identifier, which is indeed required for multi-hunk bug-fix since the identifier names may vary in two snippets of code, and search `return $x`, we will get many more similar statements. Therefore, it is evident that the statement-level similarity does not work here. On the other hand, if we use any traditional clone detection that uses a sliding window approach to detect clones using adjacent context, these repair locations will not be part of any detected clone snippets.

In order to overcome the aforementioned problem, HERCULES uses the notion of semantic context instead of the syntactic neighborhood. Specifically, HERCULES uses *reaching definition* analysis (Section IV-B3) to extract statements, within the method boundary, on which the statement at the repair location has a data-flow dependence. The extracted statements represent the semantic context of the repair location statement. This mechanism also allows HERCULES to associate a variable-sized context with a repair location. Then HERCULES performs a deeper statement-level AST analysis to determine (syntactic) similarity between the set of repair locations combined with their respective semantic contexts. For example, Figure 1 presents two contexts for two repair locations, highlighted by blue and orange color. For both hunks, the context statements are physically far from the corresponding repair location. Even more interestingly, they are interleaved. For the first hunk, the repair location is line number 230 but its reaching definition is at line number 226. For the second hunk, its reaching definition (line number 142) is 125 lines away from the repair location (line number 267). However, after HERCULES extracted the semantic context, the two resulting code snippets become similar, although not identical (declaration vs. assignment). However, HERCULES concludes that the both return statements are used in similar context, based on the AST analysis.

Weak Specification and Spurious Repair Locations. It is well known that test suites typically do not cover every program location. This issue has specific ramifications for multi-location bug fixing. For instance, in the current example, the failing test case covers the second location but not the first location. Therefore, any repair tool that solely relies on test cases for identifying repair locations, cannot generate the complete correct patch. On the other hand, if we apply program transformations in all the similar locations identified from the previous step, it may not be appropriate either. In order to find *true* evolutionary siblings, HERCULES extracts the revision history of the target repair locations and analyzes whether these lines were revised independent of one another. For our example, although the first repair location is not covered by any test case, its revision history shows it was never modified independent of the second location. This allows HERCULES to confidently apply similar changes to both locations.

Rich Repair Space. Another general limitation of G&V approaches is dealing with an enormous number of candidate patches. The number of candidate patches increases exponentially with the number of repair locations. For example, there are more than 14,000 repair expressions that are valid in each repair location. These repair expressions can be plugged into

```
univariate/BrentOptimizer.java
public class BrentOptimizer extends BaseAbstractUnivariateOptimizer {
        :
142     UnivariatePointValuePair current = new UnivariatePointValuePair(x, isMinim ? fx : -fx);
        :
226             current = new UnivariatePointValuePair(u, isMinim ? fu : -fu);
        :
                    if (checker != null) {
                        if (checker.converged(iter, previous, current)) {
230     -                       return current;
230     +                       return best(current, previous, isMinim);
                        }
                    }
        :
                    }
                }
            } else { // Default termination (Brent's criterion).
267     -               return current;
267     +               return best(current, previous, isMinim);
            }
            ++iter;
        }
```

Fig. 1: The fix for Math Defects4J ID: 24 with program context

the program transformation schemas, resulting in thousands of candidate patches. Even if we are fortunate enough to determine the correct program transformation and API call (which itself is very difficult), there can be 18 valid concrete invocations of *method call* for different combinations of parameters. For two locations, the number of candidate patches would be $18 \times 18 = 324$. These statistics illustrate the enormity of the repair space for multi-hunk bug fixes, and show why a naive approach would not work in such scenarios. In order to overcome this problem, HERCULES employs a strategy of simultaneous repair of evolutionary siblings. Furthermore, inspired by other existing approaches such as Prophet [34] and ELIXIR [40], HERCULES uses machine learning techniques to rank and prune most of the candidate patches.

IV. HERCULES

A. An Overview

Figure 2 presents the basic workflow of HERCULES. Given that there is a bug in a program (P), HERCULES takes the source code of P with its version history, a test suite (T) with at least a failing test case, and optionally a bug report, and generates a correct single-hunk or a multi-hunk patch that fixes P, in a successful run.

HERCULES works in four major steps to mutate P and eventually to generate a patch. In the first step, HERCULES uses a spectrum based fault localization (SBFL) technique to identify potential repair locations. For a given repair location, in the second step, HERCULES identifies evolutionary siblings (Definition IV.6) by leveraging reaching-definition (Section IV-B3) and version history analysis (Section IV-B4). If such evolutionary siblings are found, this step also produces the mapping between similar variables and objects among the evolutionary siblings. It is worth noting that HERCULES can also repair one-hunk bugs. Therefore, if a evolutionary sibling is not found for a particular repair location, the rest of the steps are continued with only one repair location. In the third step, HERCULES abstracts all the mapped variables and objects in all the repair locations, and instantiates repair schema simultaneously. After the patches are applied, the abstract variables are reverted back to their original variables. In the final step, HERCULES selects the Top n candidate patches for validation. HERCULES repeats these steps for each repair location by SBFL until a plausible patch is generated or timed out. In summary, HERCULES adds a novel step in the repair process that is not available in any conventional G&V approaches (Section II). We refer this step as *repair localization*, i.e., identifying the evolutionary siblings (if they exist) that require changes together to fix the bug.

B. Identification of Evolutionary Siblings

In this work, by *evolutionary siblings*, we mean the repair locations that are similar, have used in a similar context, have similar evolution history, and at least one of the repair locations has been exercised by the fault reproducing test cases. Our motivating example in Section III already showed that a traditional clone detector is not sufficient to identify evolutionary siblings. The fact is further supported by our results in Section VI-A. HERCULES works in three steps to find evolutionary siblings. Since code matching, especially at the AST level, is expensive, HERCULES lazily applies the tree similarity algorithm first at the repair location level. Then only for similar repair locations, HERCULES extracts the relevant context, and again applies the tree similarity algorithm for context. Finally, HERCULES leverages version history to find the evolutionary siblings with high confidence. In the subsequent sections, we concretely define evolutionary siblings and describe each aforementioned step in more detail.

1) Preliminaries: HERCULES represents and manipulates programs as abstract syntax trees (AST) to analyze programs

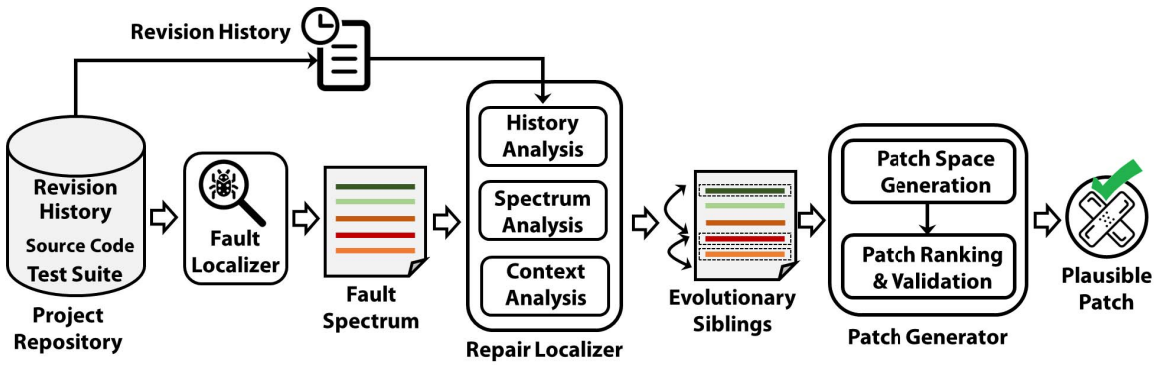

Fig. 2: Overview of HERCULES.

and to instantiate repair schemas. Although the AST representation is standard, its relevant terms and assumptions vary based on usage, and thus deserve a concrete definition in our context. In this section, we formally define all the important terms in our context.

Definition IV.1. An *abstract syntax tree (AST)* is an ordered tree (T) where each node (N) represents a program element (a method or a statement). T is a tuple of $< G, X, r, M >$ where G is a context free grammer, X is a finite set of nodes in the tree, r is the root node, and M maps r to its each children.

We also assume that for each node, there exists the following methods: $type(N)$ returns the type of a repair expression such as int or double, $kind(N)$ returns the kind of AST node such as statement or method invocation, $parent(N)$ returns the parent node of N, and $children(N)$ returns that children of N. It should be noted that the number of children nodes in N vary based on $kind(N)$, For example, a *Binary Expression* always has two children, whereas a block may have an arbitrary number of statements. Furthermore, we can also assume that all the subtrees under T are also ASTs for simplicity, e.g., an AST at statement level. However, in this case, $parent(r) \neq null$.

Definition IV.2. *Program spectra (S)* is a set of AST nodes $\{N_{s_1}, N_{s_2}, .., N_{s_n}\}$ such that $\forall N_{s_i} \in S : kind(N_{s_i}) = statement$ and exercised by the failing test cases.

Definition IV.3. A *repair location (R_l)* is an AST node, N_s, where $kind(N_s) = statement$ at a location l.

Definition IV.4. *Context of relevance (C_{R_l})* is a set of statements $\{N_{s_1}, N_{s_2}, .., N_{s_p}, R_l\}$ with respect to a repair location (R_l) such that $\forall N_{s_i} \in C_{R_1} : \Pi(N_{s_i}, R_l) = true$ where Π is a reaching-definition function.

Definition IV.5. *Edit history of a repair location (H_{R_l})* is a sequence of AST edit operations, also known as program differences, $\{D_{c_i}, D_{c_j}, ..D_{c_k}\}$, where $D_{c_i} = \Delta(R_{l,c_i}, R_{l,c_{i-1}})$ and c_i denotes the i^{th} commit in version history.

Definition IV.6. *Evolutionary siblings (E_{R_l})* with respect to a repair location R_l are a set of repair locations $\{R_l, R_{l_1}, R_{l_2}, .., R_{l_n}\}$ such that $R_l \in S$ and $\forall (R_l, R_{l_i}) \in E : \zeta(C_{R_l}, C_{R_{l_i}}) > t_1 \wedge \xi(H_{R_l}, H_{R_{l_i}}) > t_2$ where ζ and ξ are two similarity functions and t_1 and t_2 are two user defined thresholds.

2) Step-1: Identification of Similar Repair Locations: In order to find the evolutionary siblings (E_{R_l}) with respect to R_l, first we compute the similarity between R_l and each location (R_i) in program spectra S. The intuition is that if the repair locations do not match, there is no reason for analyzing their context and version history. To this end, we apply Zhang and Shasha [41] tree distance algorithm to compute the similarity between two candidate repair locations R_l and R_i. However, we introduce our own notion of similarity when comparing two AST nodes N_1 and N_2.

Kind Compatibility. N_1 and N_2 are kind compatible if $kind(N_1)$ and $kind(N_2)$ have a super-class or sub-class relationship. For example, a `variable access`, an `array access`, or a `method invocation` returning a value are of similar kind since all of them are expression type. However, a `return` statement and a `throw` statement are not kind compatible.

Type Compatibility. N_1 and N_2 are type compatible if $type(N_1)$ and $type(N_2)$ have either a super-class or sub-class relationship or satisfies the implicit type casting criteria defined by the language.

Name Similarity. We extract the name of N_1 and N_2, and use *Levenshtein Distance* algorithm to compute textual similarity.

It is worth noting that due to kind compatibility, similar but various kind of program constructs can be mapped to each other. For example, one developer may use `area=length*width` whereas another may use `area=table.getLength()*table.getWidth()`. Although a human can easily understand that `length` and `table.getLength()` represent similar data, in terms of ASTs, they are quite different. HERCULES is able to find such similarity, which is important in dealing with object oriented programming language.

Finally, we use Algorithm 1 to determine whether N_1 and

Algorithm 1 AST Node Similarity

Input: Two AST Nodes (N_1 and N_2)
Output: Similar($true/false$)?, Mapping of Elements (M)

AstNodeSimilarity (N_1, N_2)
1: $s_1 \leftarrow$ KindCompatibility(r_1, r_2)
2: **if** $s_1 == true$ **then**
3: $s_2 \leftarrow$ TypeCompatibility(r_1, r_2)
4: **if** $s_2 == true$ **then**
5: $s_3 \leftarrow$ ComputeNameSim(r_1, r_2)
6: **if** $s_3 > threshold$ **then**
7: **return** $true$;
8: **end if**
9: **end if**
10: **end if**
11: **return** $false$;

Algorithm 2 Extract Relevant Context

Input: A repair location (R_l), Source Code (SC)
Output: Set of statements representing context (C_{R_l})

ExtractRelevantContext (R_l, SC)
1: $V \leftarrow$ ExtractVariableAccesses(R_l)
2: $C_{R_l} \leftarrow \{R_l\}$
3: **for each** v in V **do**
4: $R_{l_v} \leftarrow$ ReachingDefinition(v)
5: $C_{R_l} \leftarrow C_{R_l} \cup \{R_{l_v}\}$
6: **end for**
7: **if** $|C_{R_l}|$ = 1 **then**
8: $C_{R_l} \leftarrow C_{l_r} \cup$ PreviousStatement(R_l)
9: **end if**
10: $C_{R_l} \leftarrow$ SortByLineNumber(C_{R_l})
11: **return** C_{R_l};

N_2 are similar, and eventually Zhang-Shasha algorithm to determine whether R_l and R_i are similar. In the case of similar R_l and R_i, we create a one-to-one node level mapping between the similar nodes in R_l and R_i.

3) Step-2: Determining the Relevance of Repair Locations: Step-1 certainly would remove most of the irrelevant repair locations. However, there may be still a lot of false positives since any two random statements can be similar at a statement level, especially small statements. Therefore, HERCULES enhance the analysis with program context to make sure that the repair locations are indeed similar and used in a similar context.

Significance of Program Context. Program context (i.e., surrounding code around a target location) has been used in many software engineering tasks including program repair [40]. To determine program context, the number of statements around the repair location is one of the important parameters. While a small context ($+1/-1$ statement) may not be sufficient to capture the developers' intent, a large context ($+/-10$ statements) may be difficult to generalize. To find evolutionary siblings, using adjacent program context is even further challenging since many of them are not clones in a traditional sense (as discussed in Section III).

Extraction of Program Context. In order to overcome this challenge, unlike other repair tools [40] that use a fixed size adjacent context, HERCULES uses a variable size non-contiguous relevant context (Definition IV.4). More specifically, HERCULES uses *reaching-definition* based analysis within a method boundary to extract only relevant context, even though those statements are far away from the repair location. In compiler theory, a reaching definition for a given statement (R_l) is the closest earlier statement R_i whose target variable can reach R_l without an intervening assignment.

We use Algorithm 2 to extract the relevant non-contiguous repair context for a given repair location (R_l). In words, we first extract all the variable accesses from R_l. Then for each variable access, we determine the statement that satisfies the reaching definition property based on data-flow analysis. We take a union of all the statements obtained from the analysis sorted by the line number to form the context C_{R_l}. If C_{R_l} contains only R_l, we add previous statement of R_l in C_{R_l}.

Analysis of Program Context. Once the program context for each repair location pair (R_l, R_{l_i}) is extracted, HERCULES applies the same tree matching algorithm from the previous step to determine whether the context are similar and the node mappings are still consistent. If they are similar, HERCULES marks them as potential evolutionary siblings.

4) Step-3: Revising Evolutionary Siblings Leveraging Version History: Since the accuracy of identifying evolutionary siblings is a direct impact on the repair, we further leverage version history to revise them. There may be two potential scenarios. One, some repair locations in the candidate evolutionary siblings, identified in Step-2, are independent of each other. Two, some true evolutionary siblings are not in the list due to weak test specification. Our insight is that true evolutionary siblings may have a similar evolution history, i.e., they went through similar AST operations in the past. Therefore, version history may be helpful two mitigate both problems.

Lets assume that we have three candidate evolutionary siblings ($R_{l_1}, R_{l_2}, R_{l_3}$). In order to revise the list with confidence, HERCULES first identifies all the commits ($C_1, C_2, .., C_n$) where the candidate repair locations were edited. Then it extracts the differences in each commit (before and after the changes) at the AST level (insertion, deletion, and modification). HERCULES further investigates each commit to identify if there are any other similar repair locations that have been also changed with the candidate siblings (applying Step-1). If HERCULES finds any similar repair locations (R_4, R_5), it analyzes their context as well (applying Step-2). If the results of both steps are positive, HERCULES adds such repair locations in the list of candidate evolutionary siblings. Therefore, for this example, HERCULES would get five candidate siblings. However, even if HERCULES adds some plausible siblings in this phase, it can safely discard them during validation (Section IV-D), if they introduce any regression failure,

to eventually generate a correct patch. Finally, HERCULES removes the repair locations that do not have the similar evolution history based on the edit operations. HERCULES passes the final evolutionary siblings along with the mapping of similar nodes among the repair locations to the next step for the generation of candidate patches.

It should be noted that in the whole process of finding evolutionary siblings, HERCULES applies AST matching algorithm lazily in several steps. It is indeed possible to find all the evolutionary siblings in the entire code-base and then analyze their version history. However, this approach would be inefficient.

C. Generation of Candidate Patches

In this step, HERCULES generates a single or multi-hunk patch depending on the results from the previous step. For a single repair location (R_l), HERCULES follows the traditional patch generation approach, i.e., it instantiates a select repair schema ($\Im$) with the repair expressions in scope. However, for multiple repair locations ($\Re = \{R_l, R_{l_1}, .., R_{l_n}\}$), where evolutionary siblings are involved, HERCULES instantiates the selected repair schema at an abstract level. More specifically, given a group of repair locations ($\Re$) and a mapping between similar AST nodes in $\Re$, denoted by $M(\Re)$, HERCULES abstracts all repair locations to remove the differences due to various identifier names. Then the same repair schema, $\Im$ is instantiated at the abstract level at each repair location simultaneously. Once the transformation, $\Im$ is applied, HERCULES generates the concrete candidate patches by reverting the original variables using $M(\Re)$. Therefore, the repair space generated by HERCULES may have both single or multi-hunk patches. It is worth noting that, due to simultaneous repair schema instantiation, HERCULES keeps the repair space comparable to the repair tools that only focuses on generating single-hunk patch.

D. Ranking of Candidate Patches and Validation

Once the candidate patches are generated, HERCULES can use any ranking models (heuristic based or machine learning based) proposed in the existing repair tools to rank the candidate patches, and selects Top N candidate patches for validation, one at a time. During the validation phase for each candidate (single or multi-hunk) patch, HERCULES first runs the failing tests, and if they pass, HERCULES runs the regression tests. If the regression test suite pass, HERCULES stops and reports that patch as a final patch.

V. EXPERIMENTAL SETUP

A. Implementation

HERCULES is a G&V-style repair tool implemented in Java. It includes a spectrum-based fault localizer as well as a source code and version history analyzer. HERCULES also implements a mechanism for applying repair schemas simultaneously at several locations, to effect multi-hunk repairs. Inspired by previous well-known program repair tools [22, 29, 40], HERCULES includes repair schemas such as checking null pointer check, changing and inserting method invocation, changing and inserting *if* conditions, and so on. Further, it incorporates a machine learning based patch ranking model similar to ELIXIR [40]. For the details of each of the repair schemas and the ranking model of candidate patches, the reader is referred to the corresponding papers [22, 29, 40]. We also implement several baseline versions of HERCULES to demonstrate the effectiveness of its various components (Section VI-D).

B. Dataset

We used the popular Defects4J dataset [6] to evaluate HERCULES, specifically the five subjects: Math, Lang, Chart, Time, and Closure.

C. Training HERCULES

Since HERCULES uses a machine learning technique to rank and prune candidate patches for validation, we train HERCULES with real-world bugs. In order to train HERCULES, we used another publicly available real-world bug dataset, Bugs.jar. There are 1,158 real-world bugs in Bugs.jar taken from eight well-known large Apache projects. Among them, Apache Commons Math is common to both Defects4J and Bugs.jar [7]. Therefore, we removed Apache Commons Math from our training set, to keep the training and testing datasets mutually exclusive.

D. Research Questions

RQ1: How effective is HERCULES for repair localization through evolutionary sibling detection compared to traditional clone detection?

RQ2: How effectively does HERCULES generalize the traditional single-hunk repair strategy of APR tools to perform both single-hunk as well as multi-hunk repair?

RQ3: How effective is HERCULES in repairing programs compared to state-of-the-art program repair tools?

RQ4: What is the contribution of various components in HERCULES to its overall bug-fixing capability?

E. Experimental Configurations

We ran all the experiments on a cluster of Virtual Machines (VM), where each VM was configured to have double core 3.6GHz processor and 4GB memory. We used Ubuntu 16.04 LTS operating system and Java 7. There are several configuration parameters in HERCULES as well. HERCULES used threshold value 0.8 for determining tree similarity, iterated through Top 200 repair locations and selected 50 candidate patches per repair schema. We set a time out of 5 hours following the recently introduced repair tool, SimFix [27].

VI. RESULTS

A. RQ1: Effectiveness of HERCULES for Repair Localization vs. a Traditional Clone Detector

Motivation. The main contribution of HERCULES is that it enables fixing a specific but prominent class of multi-hunk bugs by accurate repair localization, which is in turn achieved through the detection of evolutionary siblings. Therefore, it is

TABLE I: Effectiveness of HERCULES and Deckard for Repair Localization

Patch-Type	Bugs	Approach	Correct	Proportion
Single-hunk	130	HERCULES	114	88%
		Deckard	38	29%
Multi-hunk	24	HERCULES	15	63%
		Deckard	2	8%
Total	154	HERCULES	129	84%
		Deckard	40	26%

important to evaluate how effective HERCULES is in detecting evolutionary siblings, especially compared to a traditional clone detection tool since if a clone detector can detect all the evolutionary siblings accurately we do not need any sophisticated repair localization.

Experiment. In order to make a meaningful comparison, we first identify all the relevant bugs in Defects4J for this experiment. Here, by *relevant bugs* we mean all the bugs that involve either only a single-hunk patch or ones with a multi-hunk patch but similar edits in all hunks. The rest of the bugs are, by definition, out of scope and will only add noise to the experimental results. This gave us 154 bugs in total, 130 single-hunk bugs and 24 multi-hunk bugs. Recall from Section IV-B that, for a given buggy location, the objective of repair localization is to identify all the relevant locations where similar changes are required to repair the bug correctly and completely. Therefore, in an ideal case, if we point a clone detector or HERCULES to an actual buggy line (*i.e.,* an input location), it should return only one repair location (the input location itself) for a single-hunk bug and all the n relevant repair locations (including the input location) for an n-hunk bug. To this end, we ran the *repair localization* component of HERCULES and an established clone detector, Deckard [13] on all the 154 bugs in Defects4J with respect to their actual buggy locations (or the one with the highest fault-localization rank, for a multi-hunk bug). For Deckard, we set the minimum number of tokens to 10 (approximately two lines of code) following [16].

Results. From Table I, we see that HERCULES identified the repair locations correctly for 84% (129 out of 154) bugs whereas Deckard found the correct locations only for 26% (40 out of 154) bugs. On closer inspection, we see that Deckard performed even poorer on multi-hunk bugs, which is the main target of repair localization, than single-hunk bugs. It detected correct repair locations for only 2 bugs, while HERCULES correctly localized 15 out of 24 (63%) of these bugs. Even for single-hunk bugs, Deckard correctly localized only 29% (38/130) of the instances compared to HERCULES's 88%. In both single-hunk and multi-hunk instances Deckard produced false positives as well as false-negatives. The reason is simply that many of the target repair locations are not clones in a traditional sense (like our motivating example in Figure 1), and beyond the scope of Deckard's purely syntactic, fixed-context clone detection. By contrast, HERCULES's specialized evolutionary sibling analysis, which combines several orthgonal sources of information, is far more accurate, and hence indispensible, for the application at hand, *i.e.,* multi-hunk program repair.

TABLE II: Effectiveness of HERCULES (Correct/Incorrect)

Subject	Math	Lang	Time	Chart	Closure	Total
HERCULES	22/7	10/4	3/1	5/3	6/2	46/17
HERCULES-SH	13/7	8/4	2/1	4/3	5/2	32/17

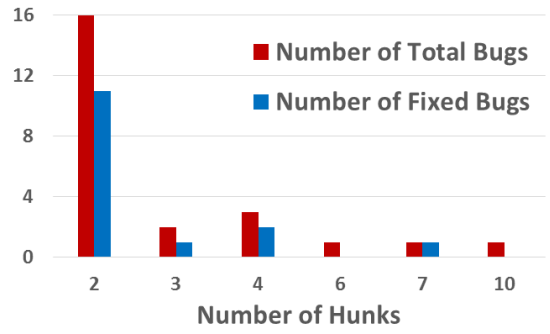

Fig. 3: Distribution of Multi-Hunk Bugs in Our Study

B. RQ2: Effectiveness of HERCULES *in Generalizing Single-hunk Repair to Combined Single-hunk and Multi-hunk Repair*

Motivation. In RQ1, we demonstrated that HERCULES is effective in detecting evolutionary siblings. The current experiment evaluates how well this repair localization translates into the end goal of producing correct patches, not just for multi-hunk instances but also single-hunk patches.

Experiment. We first create a baseline, named HERCULES-SH that has all the repair schemas in HERCULES but only performs single hunk repair. Then we ran both HERCULES and HERCULES-SH on all the 154 bugs described in RQ1.

Results. As presented in Table II, HERCULES generates 46 correct patches and 17 incorrect patches while HERCULES-SH generates 32 correct patches and 17 incorrect patches as well. A deeper look into the results reveals that HERCULES fixed 31 single-hunk bugs and 15 multi-hunk bugs. By comparing the results with HERCULES-SH, we observe that HERCULES lost one single-hunk bug due to incorrect simultaneous repair. Surprisingly, HERCULES does not generate any additional incorrect patches due to multi-hunk repair. The reason is that in multi-hunk repair, since program changes happen in two or more repair locations, the probability of detecting regression bugs by the test suite increases significantly. Even if a test case detects regression in one repair location out of n, the whole candidate patch is discarded.

An analysis of the distribution of multi-hunk bugs and patches, in terms of number of hunks, offers some interesting observations. As presented in Figure 3, HERCULES fixed a variety of multi-hunk bugs, ranging in size from 2 hunks to as large as 7 hunks.

TABLE III: Statistics of Patch Generation by Various Techniques (Correct/Incorrect)

Subject	Math	Lang	Time	Chart	Closure	Total
HERCULES	21/7	10/3	3/2	6/3	6/2	46/17
SimFix	14/12	9/4	1/0	4/4	6/2	34/22
CapGen	13/-	5/-	0/-	4/-	0/-	22/-
JAID	1/-	1/-	0/-	2/-	5/-	9/-
ELIXIR	12/7	8/4	2/1	4/3	0/-	26/15
ssFix	10/16	5/7	0/4	3/4	2/9	20/40
ACS	12/4	3/1	1/0	2/0	0/-	18/5

C. RQ3: Effectiveness of HERCULES vs. State-of-the-art Program Repair Tools

Motivation. In RQ2, we demonstrated that HERCULES is effective in fixing multi-hunk bugs as well as single-hunk bugs. In this section, we evaluate the effectiveness of HERCULES with respect to the state-of-the-art program repair approaches.

Experiment. We choose six of the most recent (Java) APR tools: SimFix [27], CapGen [26], JAID [42], ELIXIR [40], ssFix [30], and ACS [29] as representative of the state of the art. Since all these tools have already been evaluated on Defects4J, we simply take the results from the respective papers.

Results. Table III presents the number of correct and incorrect patches generated by various tools. From the results, we see that HERCULES generated 46 correct patches, which is the most among all tools. It fixes 12 more patches than the next best tool, SimFix.

Since SimFix and ACS have also the ability to fix multi-hunk bugs, we investigated the results at the level of individual bugs. We found that HERCULES fixes 9 unique multi-hunk bugs that SimFix and ACS cannot fix. Furthermore, if we consider all the bugs, HERCULES fixed 11 unique bugs that no existing approaches could fix. Overall the results demonstrate that HERCULES advances the state of the art significantly.

D. RQ4: Contribution of Various Components of HERCULES

Motivation. Evaluations in the previous RQs demonstrate that HERCULES overall outperforms existing tools. The next experiment evaluates the contribution of various features of HERCULES to its overall bug-fixing capability.

Experiment. In order to investigate the contribution of various components, we create three versions of HERCULES.

- HERCULES-FIXEDCONTEXT uses a fixed number of lines w before and after the buggy location as its context, rather than the semantic context. For this version we experimented with $w = \{1, 2, 3\}$ and reported the best results.
- HERCULES-MINUSHISTORY ignores the version history in repair.
- HERCULES-INCR implements an incremental repair strategy where there is no notion of evolutionary siblings. Each location is fixed independently, one after another, as long as each successive repair decreases the number of failing tests. This strategy broadly mimics the one implemented in SimFix [27] and ACS [29].

TABLE IV: Comparison among variants of HERCULES

Approach	Number of Correct Patches
HERCULES	46
HERCULES-FIXEDCONTEXT	35
HERCULES-MINUSHISTORY	40
HERCULES-INCR	37

We run this experiment only for the 46 bugs that HERCULES correctly fixed.

Results. Table IV presents the aggregated results in terms of number of correct patches. From the results, we observe that when we used a fixed size context, HERCULES lost 11 bugs. Similarly, version history was crucial in fixing six bugs correctly. Finally, the results further show that when HERCULES does not leverage the information of evolutionary siblings, *i.e.,* works in an incremental fashion, it cannot fix 9 bugs. In summary, the results demonstrate that all the features of HERCULES contributed in fixing multi-hunk bugs.

VII. LIMITATIONS & THREATS TO VALIDITY

Scope of multi-hunk repairs. Our current technique addresses only a specific class of multi-hunk repairs, namely ones with substantially similar patches for each hunk. While this does boost the successful repairs by almost 50% compared to the baseline tool, future research needs to address other classes of multi-hunk bugs, the vast majority of which are still out of scope for HERCULES or any other APR tool.

Accuracy of version history analysis. Our version history analysis can be impacted by noise in the revision history introduced by major structural changes to the repository, such as to the directory or package structure, or other systemic refactoring changes. We use several heuristics to compensate for such disruptions and manually inspected the history of a few sampled bug instances to verify the accuracy of the heuristics. However, we cannot guarantee the soundness of the analysis.

Generalizability of the results. Our evaluation was only carried out on the Defects4J dataset, which is a widely used benchmark for program repair research. However, the dataset's 5 subject systems cannot capture the wide variety of Java applications and their bugs. Further validation of this technique on other subjects should necessarily be done in future. Further, our current repair results depend on the capabilities of the baseline repair tool, on which our multi-hunk repair technique is implemented. Although our baseline APR tool is quite competitive with the state of the art, using a different or improved APR tool, such as SimFix [27] for example, could change or improve the results. Lastly, our technique has been instantiated for Java program repair but could, in principle, be applied to C/C++ program repair as well. But its efficacy in that setting remains to be investigated.

VIII. RELATED WORK

Automatic program repair. A decade of research has generated a rich body of work on program repair, summarized in two excellent recent surveys [1, 2]. With the notable

exception of Angelix [5] APR research so far does not target multi-location bugs, which is our main focus. However, APR research most related to HERCULES's approach can be classified into techniques that: (1) mine existing code or patches for repair fragments, *e.g.,* variables, expressions, statements, or complete code snippets, *etc.* , (2) learn abstract repair spaces, *e.g.,* program transformations from existing patches, and (3) can produce multi-hunk patches.

APR - Mining repair fragments: GenProg [28], which pioneered this area, uses genetic search on a space of repair mutations formed by code snippets copied from elsewhere in the program. RSRepair [38] and AE [39] follow GenProg, using random and deterministic search respectively, instead. μSCALPEL [43] transplants code snippets mined from a donor application to repair bugs in a donee application. CodePhage [44] also performs transplantation but targets missing if-condition related bugs. ACS [29] performs if condition repairs using predicates mined from Github. SearchRepair [45] mines repair fragments using a semantic search, based on SMT formulas and constraint solving. ssFix [30] performs the same search using a general purpose search engine. SimFix [27] also searches for a donor snippet, but further uses a mined space of abstract schemas to prune the search. Prophet [34] and Elixir [40] do not directly mine repair artifacts but rather use a corpus of existing patches to train a classifier, which is then used to rank the space of concrete patches. As a whole the above techniques search for compatible code fragments (or features thereof) to contribute to the repair of the bug at hand *i.e.,* a donor-donee relationship. By contrast, HERCULES's search for "similar" code is used to find a set of evolutionary siblings that can be repaired concurrently.

APR - Learning abstract repair spaces: PAR [22] first used this approach, defining its repair space using a set of 10 specialized repair templates manually derived from human-written patches. Relifix [23] uses specialized repair schemas customized for software regression errors. History-driven repair [46] prioritizes its pool of candidate repairs based on the frequency of occurrence of the repair in a corpus of past (human-written) patches. Tan et al. [24] propose a blacklist abstract repair space defined by a set of anti-patterns, to curb the generation of incorrect patches. Genesis [25] automatically extracts a set of repair schemas for specific classes of bugs by solving an optimization problem on the set of previous patches for each bug class. CapGen [26] mines a set of 30 frequently occurring AST-level transformations from a corpus of previous patches, and uses them for repair. Our contribution is fundamentally orthogonal to the above body of work in that our identification of evolutionary siblings seeks to identify *where* the repair should be performed while the above informs *how* the repair at a *given location* should be performed.

APR - Multi-hunk repair: So far Angelix [5] is the only APR tool to specifically target multi-hunk patches. It generates a symbolic oracle for potential changes to the top 'k' locations given by fault localization and then independently synthesizes patches for each location from the oracle. Although not specifically described in [27, 29] the implementations of ACS [29] and SimFix [27] are in fact capable of performing simple instances of multi-hunk repair. Specifically, when there are distinct test-cases separately implicating each of many bug locations, the tool independently repairs these bug locations, one after another, using the *number* of passing tests as a progress metric. Our key contribution is that we employ sibling relationships between locations to significantly cut down the search space by repairing locations simultaneously, localize the repair more accurately, and can even repair locations not directly implicated by the test spectrum.

Code Clone Detection. Our work is broadly inspired by research on code clone detection and extraction of code clone genealogies. Deckard [13], NiCad [14], and CCFinder [12] are some of the popular clone detectors. Kim et al. [8] were the first to extracted and analyze clone genealogies across multiple revisions of a software system to understand the evolution of code clones. However, as demonstrated in RQ3, the evolutionary siblings that form the basis of our approach may or may not be traditional code clones.

Systematic Edits. Our approach is also inspired by research on systematic edits [15, 47], with LASE [16] and RASE [17] being the most advanced tools in this area. The key idea in this research is to learn a common abstract edit script from a few examples instances of it, which is then replicated at target locations in the code, also identified using the learned edit script. HERCULES also shares the notion of similar edits at multiple locations. However, we do not rely on examples for identifying sibling locations.

IX. CONCLUSION

Automatic program repair techniques have made significant advances over the past decade. However, practical deployment remains an elusive goal. One of the significant obstacles to achieving this goal, is the inability of current APR techniques to produce multi-hunk patches. In this work, we presented a novel APR technique that generalizes single-hunk repair to encompass a specific but significant class of multi-hunk repair problems, namely ones that require applying a substantially similar patch at a number of locations. We term such sets of repair locations as evolutionary siblings – similar looking code, instantiated in similar contexts, that are expected to undergo similar changes over time. We proposed a novel analysis to accurately identify a set of evolutionary siblings, for a given bug. This analysis combines three orthogonal sources of information, namely, the test-suite spectrum, a novel code similarity analysis that compares both syntactic and semantic features, and the revision history of the project. We implemented this technique in a tool HERCULES and demonstrated that it is able to correctly fix 46 bugs in the Defects4J dataset, the highest of any individual APR technique to date. This includes 15 multi-hunk bugs, and 11 bugs which have not been fixed by any other technique so far. We see this contribution as a small but important step on the road to achieving practical deployment of APR tools.

REFERENCES

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[2] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018.

[3] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[4] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 913–923.

[5] 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*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701.

[6] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[7] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs. jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*. ACM, 2018, pp. 10–13.

[8] M. Kim and D. Notkin, "Using a clone genealogy extractor for understanding and supporting evolution of code clones," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[9] C. K. Roy and J. R. Cordy, "An empirical study of function clones in open source software," in *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008, pp. 81–90.

[10] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 158–167.

[11] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 485–495.

[12] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[14] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ser. ICPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 219–220.

[15] 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*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 329–342.

[16] 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*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 502–511.

[17] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 392–402.

[18] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen?s School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[19] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 100–109.

[20] J. F. Islam, M. Mondal, and C. K. Roy, "Bug replication in code clones: An empirical study," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1. IEEE, 2016, pp. 68–78.

[21] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 306–317.

[22] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 802–811.

[23] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 471–482.

[24] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 727–738.

[25] 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*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 727–739.

[26] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 1–11.

[27] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 298–309.

[28] 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*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 3–13.

[29] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 416–426.

[30] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 660–670.

[31] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 24–36.

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

[33] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the defects4j dataset," *CoRR*, vol. abs/1505.07002, 2015. [Online]. Available: http://arxiv.org/abs/1505.07002

[34] 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*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 298–312.

[35] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[36] T. Janssen, R. Abreu, and A. J. van Gemund, "Zoltar: a spectrum-based fault localization tool," in *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*. New York, NY, USA: ACM, 2009, pp. 23–30.

[37] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[38] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265.

[39] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. Piscataway, NJ, USA: IEEE Press, Nov 2013, pp. 356–366.

[40] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 648–659.

[41] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[42] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 2017, pp. 637–647.

[43] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 257–269.

[44] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 43–54.

[45] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 295–306.

[46] X. B. D. Le, D. Lo, and C. Le Goues, "History Driven Program Repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. Piscataway, NJ, USA: IEEE Press, March 2016, pp. 213–224.

[47] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.