Program Repair Using Code Repositories

Qi Xin
Brown University
qx5@cs.brown.edu

Steven P. Reiss
Brown University
spr@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

***Abstract*—We present our system RepoRep for repairing faulty programs at the source code level using large code repositories. It has four key steps: fault localization, code search, renaming, and patch generation. Fault localization identifies a program's faulty locations; code search uses keywords describing the faulty code to find candidate code for repair in the repositories; renaming unifies the candidates with the faulty code; and patch generation uses the unified candidates to generate potential fixes. We have implemented RepoRep for automatically fixing Java programs, using GitHub as the code repository where the only manual input is the keywords. We demonstrate the effectiveness of our approach by running RepoRep to repair 27 buggy programs derived from Fry and Weimer [1] and by comparing it to GenProg and Nopol using the same programs. RepoRep can successfully repair 19 of these with correct patches while GenProg and Nopol can each make correct repairs for less than 5.**

I. Introduction

Fixing bugs is laborious. Automatic fault localization techniques [2], [3], [4], [5], [6] are capable of localizing the suspicious parts of a program using the program's executions against test cases. However, given the faulty code portions, it is still the programmer's job to fix the bugs.

Program repair tools try to do this automatically, generally at the source level. For instance, GenProg [7], [8] takes a faulty program and a set of test cases as inputs, creates an initial set of program variants, and uses genetic programming to evolve a set of variants in search of a good patch. GenProg generates variants using the same program's source. In doing so, it implicitly assumes that the correct behavior is somewhere within the same program.

While the correct behavior may appear elsewhere in the same program, as noted by [9], [10], it may also appear in other programs. Our approach aims to find any of those programs for bug-fixing. We assume the fix to the bug is relatively small compared to the program and the code representing the correct implementation of the program could be obtained via searching for code that is similar (but not identical) to the buggy code. While code similarity could in general be defined in multiple forms, in this paper, we focus on code syntactic similarity (see Section III-B). The study [11] showed that there is significant syntactic redundancy within a medium-sized (6,000 projects, 420 million LOC) code repository. This should be more significant for today's code repositories which are huge and are rapidly growing. As of 2016, Open Hub [12] claims to contain 31 billion LOC and GitHub claims to have 35 million repositories [13]. Therefore, for a reasonable code chunk size, we believe it is possible to find syntax-similar chunks in the repository that are solving a similar problem and hence could represent a correct implementation of the buggy code.

Our research makes use of this assumption. Our system RepoRep starts with a buggy but runnable Java program and a set of test cases. It employs automatic fault localization to identify a code chunk containing the buggy code based on the test cases. The user supplies keywords describing that code chunk. RepoRep then uses code search to get candidate code from GitHub, matches the candidates to the buggy code chunk and finally generates patches based on the candidates.

To allow useful candidates to be effectively retrieved, we believe the size of the code chunk is critical. A chunk that is too small is often lack of enough context. However, a chunk that is too large is likely to be unique and could lead to many false positives being found that match to the irrelevant parts of code. In this paper, we assume the code chunk is a *method*. We leave the study of a better form of code chunk for future work.

The goal of code search is to use repository code to repair the buggy methods. Current techniques are limited by the existing code in the program and their reliance on test cases. Although systems like GenProg, RSRepair [14] and AE [15] are able to propose a number of patches, according to [16], the majority of those patches are not correct: GenProg can only fix 2 of the 105 considered bugs with correct patches. RSRepair [14] and AE [15] have also poor patching performances. The main problem is that the generated patches are artificial and overfitted [17]. By considering code outside the project that does a similar task and by assuming this code does the task correctly, we hope to extend the fixing capabilities and performance of automatic bug repair. There are other techniques [18], [19] similar to ours that make use of code in the repository that is semantically similar, or more strictly, semantically correct for bug-fixing. Compared to these techniques, RepoRep's syntactic code search is more efficient. It is able to utilize any code (of the same language) in the repository without having to compile or execute the code.

This paper describes RepoRep showing the feasibility and promise of our approach. Our contributions are:

1) We present a novel system RepoRep for program repair using today's large external code repositories.
2) We show how to identify code in a repository that can be a candidate for repair.
3) We create heuristics and algorithms for unifying the names of the candidate and target code and for generating

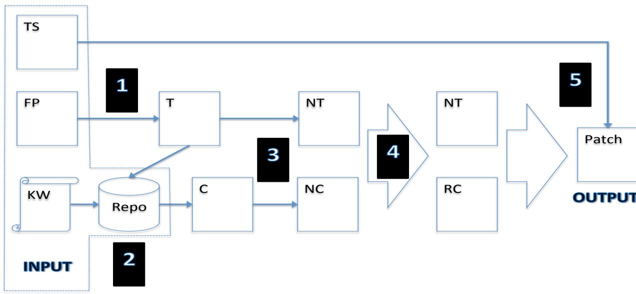

Fig. 1. Approach Overview. TS: Test Suite. FP: Faulty Program. KW: Keywords. Repo: Code Repository. T: Target. C: Candidate. NT: Normalized Target. NC: Normalized Candidate. RC: Renamed Candidate. Step 1: Fault Localization. Step 2: Code Search. Step 3: Normalization. Step 4: Renaming. Step 5: Patch Generation.

```
1 private static int indexOfSmallest(
2 int startIndex, Comparable[] a,
3 int numberUsed) {
4   Comparable min = a[startIndex];
5   int indexOfMin = startIndex;
6   int index;
7   for (index = startIndex + 1;
8        index <= numberUsed; index++)
9     //{ Block inserted by normalization
10    if (a[index].compareTo(min) < 0) {
11      min = a[index];
12      indexOfMin = index;
13    }
14    //}
15  return indexOfMin;
16 }
```

Fig. 2. An Example of Target Code.

patches to the buggy code using the candidates.

4) We demonstrate through experiments that our techniques can work and can work well.

Section II provides an overview of RepoRep's repairing process. Section III describes the design and implementation of RepoRep in detail. Section IV describes the experiment used to validate our approach and provides an analysis of the results. Section V describes related work.

II. Overview

An overview of RepoRep is presented in Figure 1. For bug-fixing, RepoRep requires as input a faulty program and a set of test cases, some of which the program fails. It also requires user-provided keywords to access GitHub for code search. What RepoRep outputs is a patch (if any exists) that can pass the test cases. In doing so, the design of RepoRep needs to address several problems.

First RepoRep uses fault localization techniques to find the buggy method as the fixing target (Step 1). Of the numerous automated techniques [2], [3], [4], [5], [6] available, it uses Tarantula [5] for this purpose. The user then supplies keywords describing that code. RepoRep uses these keywords to search GitHub (Section III-B), yielding files representing Java classes. For each file, it extracts all the methods and uses them as potential candidates (Step 2). To be useful, RepoRep needs to identify candidates that are likely to fix the bug. We assume the fix is relatively small and a useful candidate has similar context to the buggy target code. The first problem is ranking these potential candidates according to their similarity (defined in Section III-B) to the target.

When using a candidate to fix the target, RepoRep needs to do some unifications between them. It first needs to eliminate purely syntactic differences (Step 3), and the second problem is normalizing the code to account for this. Next RepoRep needs to map the names in the candidate to match those in the target (Step 4). There can be a large number of such mappings, and the third problem to be addressed is finding the most appropriate ones.

Once a name-matched candidate is found, RepoRep uses the differences between it and the target to generate patches and uses the test suite to validate the patches (Step 5). If the differences are minimal, this is easy. Otherwise, many potential patches could result. The fourth problem is how to generate reasonable patches based on the differences.

To determine how to best address these problems, we selected a set of 20 examples to train RepoRep. These were manually collected from the IntroClass bugs [20] (selected programs were rewritten in Java), StackOverflow [21], and Defects4j [22] and varied in complexity, scope, and domain. They were independent of any examples we later used for evaluation. We used these examples to determine appropriate algorithms and heuristics for each of the above problems. Where the algorithms involved weights and probabilities, we used the examples to determine appropriate values (Section III-C2).

Figure 2 shows the paper's running example. `indexOfSmallest` is an auxiliary method for selection sort. It returns the index of the smallest value of `a` from index `startIndex` to `numberUsed`. The bug is in the conditional: `index <= numberUsed` should be `index < numberUsed`.

III. Methodology

We now elaborate on these stages of RepoRep.

A. Fault Localization

The first stage identifies the suspicious method of the program as the target to fix. RepoRep first runs the program with Jacoco [23] against the test cases to get code coverage and uses Tarantula [5] to compute the suspiciousness of each method. The result is a list of methods sorted by their suspiciousness. RepoRep selects all methods above a certain threshold (half of the maximum) as the targets, assuming one of them contains the bug. It then executes the subsequent steps independently on each selected method.

B. Generating Candidates

The next stage finds candidates for patching. The ideal candidate code is essentially the same as the original code except for the necessary bug fix. Assuming that the fix is small

relative to the size of the original code, this means finding code similar (but not identical) to the target.

Given the identified faulty method M and a set of user-provided keywords K, we define that a candidate method M' is similar to M if (a) the pure program text of M' contains some keyword $k \in K$ and (b) the parameterized program text of M' is similar to M according to the Levenshtein Similarity. (Program parameterization is explained at the end of this section.) Condition (a) makes sure the candidate method is conceptually similar to the target in terms of the keywords. Condition (b) makes sure the candidate method is syntactically similar to the target in terms of the program structures.

Repositories like GitHub offer a keyword search front end returning files containing those keywords in a priority order. RepoRep asks a user to choose keywords describing the target domain (which is the only manual step in RepoRep's overall process) and accesses GitHub through the keywords to obtain an ordered set of candidate methods that satisfy condition (a). RepoRep is inevitably sensitive to the selection of keywords as discussed in Section IV-B3.

Given the keywords, RepoRep accesses the top N pages of search results from GitHub. (We choose $N = 20$ for experiments.) It creates a potential candidate solution for each method in each returned file. This candidate includes the method as well as the surrounding class since field and helper methods might be required.

To further select syntactically similar candidates, RepoRep compares the parameterized program texts between the target and each of candidates. In order to create the parameterized program texts, RepoRep first normalizes the target and each of the candidates to eliminate syntactic differences. This is done using semantic-preseving tranformations. While many potential normalization rules exist, RepoRep currently only uses the simple transformation that replaces a single statement with a block containing just that statement. Our initial experiments found that this was often sufficient. If needed, RepoRep can be configured with more complex transformations. After normalizing the code in Figure 2, the `if` statement has been inserted into a block, which now becomes the loop's body.

RepoRep next identifies the most likely candidates from the set of potential ones by prioritizing them based on their code similarity to the target. Since this is akin to finding code clones [24], [25], [26], [27], [28] of the target, RepoRep takes a similar approach using the ideas in CCFinder [24]. Given a normalized target and candidate, RepoRep parameterizes their program texts and represents their token sequences as strings called the *code patterns*. These strings capture the underlying program structures while omitting low-level details such as names and literals. Clone similarity then reduces to the evaluation of string similarity, for which RepoRep uses *Levenshtein Similarity*. The candidates are ranked using this metric.

RepoRep uses a slightly modified version of CCFinder's patterns that our initial experiments showed yielded more accurate results. Instead of using a consistent symbol `$p$` for all the name tokens, RepoRep replaces the tokens of type

```
1 public static void sortAnything(
2 Sortable items[], int numOfItems) {
3   Sortable temp;
4   int indexSmallest; int index1; int index2;
5   for (index1 = 0;
6        index1 < numOfItems - 1; index1++) {
7     indexSmallest = index1;
8     for (index2 = index1 + 1;
9          index2 < numOfItems; index2++) {
10      if (items[index2].lessThan(
11          items[indexSmallest])) {
12        indexSmallest = index2;
13      }
14    }
15    temp = items[index1];
16    items[index1] = items[indexSmallest];
17    items[indexSmallest] = temp;
18  }
19 }
```

Fig. 3. An Example of Candidate Code (Normalized)

names with the symbol `$t$`, method names with `$m$`, and variable names and literals with the `$p$`. All other types of tokens are kept intact. Thus, for example, the generated code pattern of "`Comparable min=a[startIndex];`" from Figure 2 becomes "`$t$$p$=$p$[$p$];`". RepoRep considers the 100 top-ranked candidates for the subsequent steps. This value was chosen based on our initial experiments to balance candidate variety and repairing efficiency. Figure 3 shows a candidate example chosen this way by RepoRep using the keyword "smallest index".

C. Renaming

Next RepoRep renames the top candidates so that they use the same names as the target. This treats each candidate independently (and, in the implementation, in parallel). For each candidate, it generates matchings that map names in the target to those in that candidate. Using a branch-and-bound algorithm it generates the top three matchings for each candidate. Our renaming experiments (Section III-C2) with the training data show that a candidate that has the potential to fix a target is often close to it and there are only a few mappings that make sense; hence it was enough for RepoRep to only consider the top three mappings.

1) Create the Name Mappings: We define a *Name Mapping* as a mapping that takes each identifier in the candidate and maps it either to an identifier in the target or to itself such that:

- **Consistency.** Variable identifiers are mapped to variable identifiers, types to types, methods to methods.
- **α-Conversion.** Identifiers with the same binding in the candidate are mapped to the same identifier.
- **Compatibility.** Candidate identifiers are only mapped if their underlying data types are compatible.

We define *Variable Mapping*, *Type Mapping* and *Method Mapping* as different types of name mappings that map the variable identifiers, data types and method identifiers respectively in the candidate subject to the above constraints. We define a

Renaming Element (*RE*) as an instance of a bound identifier. For example, in Figure 2, the variable `index` is an RE. With the definition of RE, the mappings can be thought of as mapping REs in the candidate either to themselves or to REs in the target via α-conversion. We do not consider mappings that are either inconsistent or incompatible.

Variables are the most prevalent identifiers in most code. For this reason, RepoRep first finds the best variable mappings and then extend each of these to name mappings using the best type and method mappings.

RepoRep determines the best variable mappings by generating a score for each possible mapping and then choosing the mappings with the highest score. The score is meant to reflect the quality and appropriateness of the mapping. It is computed by summing the scores for each mapping from a candidate variable RE (cVRE) to a target variable RE (tVRE) or to itself. RepoRep uses a branch-and-bound algorithm to make this practical.

The heuristic score for a mapping from cVRE to tVRE is based on three things: the *declared types*, the *names*, and, most important, the *def-use instances* which reflect the similarity of the contexts in which the names are used. RepoRep first looks at their declared types to determine whether the REs are compatible. The REs are incompatible iff the declared types are incompatible library types (e.g., `String` v.s. `List`) or incompatible primitive types [1] (e.g., `int` v.s. `boolean`). If types are incompatible, the mapping score is just zero (i.e., cVRE and tVRE are not matched), irrespective of how the names or the def-use instances are scored. (Note that two REs are compatible if any of their declared types is non-library. This is because the types can potentially be renamed to be the same.) For compatible types, two variables of the primitive and library types are more likely to match. RepoRep uses the score *sdecltype* (Section III-C2) for compatible primitive types and compatible library types. (For array types, RepoRep uses the compatibility of their base types.)

Two variables with the same name are likely to match, especially if that name is long, since they are likely used for similar purposes. RepoRep uses a name score *slname* if the names are longer than one character and equal, and a score of *ssname* if the names are the same one character name. It takes this conservative approach since names that are similar do not necessarily have similar meanings.

Analyzing and comparing the def-use instances, which represent the contexts of names, is more complex. We define a def-use instance of a variable RE to be an instance of how the RE is defined or used in a context. For example, there are seven def-use instances of the variable RE `index` in Figure 2, one of which is in line 11. A def-use instance encodes the contexts by the *context locations* (*ctxt_locs*) and the *context expressions* (*ctxt_exps*). A context corresponds to an predecessor of the RE instance in the AST where the immediate context is the instance's parent. *ctxt_locs* is a list of RepoRep-defined types of locations in which an RE appears. For example, *ctxt_locs* for `index` in line 11, Figure 2 is {*if-then-body, for-body, method-body*}. *ctxt_exps* is a list of parametered expressions in which an RE appears. For example, *ctxt_exps* for the same instance of `index` is a list of parametered expressions of {`a[index],min=a[index],if(a[index].compareTo(...){...},for(index=startIndex+1;...){...}`}. RepoRep associates each def-use instance of cVRE with a def-use instance of tVRE. It scores all possible such associations and uses the best score as the overall def-use instance matching score. The score for a particular def-use instance is the sum of its location and expression scores. The def-use instance and the matching for type RE and method RE are likewise defined.

[1]We define *byte, short, int, float, double* and *long* types are compatible to each other. *char* and *boolean* are only compatible to themselves.

TABLE I
CONTEXT LOCATIONS

Normal	Special
class-field	method-parameter
class-method	catch-clause-parameter
method-body	enhanced-for-statement-parameter
catch-clause-body	enhanced-for-statement-expression
if-then-branch	loop-condition
if-else-branch	loop-body
labeled-body	for-initializer
switch-body	for-updater
synchronized-body	labeled-label
try-body	switch-expression
finally-body	synchronized-expression

Now we explain the RepoRep-defined location types and the parametered expressions in detail. To understand the types of locations in which names appear, we identify using our training data twenty-two different context locations shown in table I to be used by RepoRep. We categorize these as either *normal* or *special*, where special indicates that the context is probably more important (and thus weighted more) in identifying whether two identifiers represent the same name. Each use of an RE has an associated list of context locations where the first item is its immediate location, the second is a context location that it is nested in, etc. Two uses are scored by looking at their context location lists, and adding scores for each location in those lists until the lists differ. Matching special locations are scored as *ssloc* and matching normal locations as *snloc*.

Let us take an illustrative example of matching two def-use instances of REs: `index` (line 12 of Figure 2) and `index2` (line 12 of Figure 3). The context locations for `index` are *if-then-branch*, *loop-body* and *method-body*. The context locations for `index2` are *if-then-branch*, *loop-body*, *loop-body* and *method-body*. RepoRep considers *if-then-branch* (normal) and *loop-body* (special) as their common context locations. The location score is $snctxtloc + ssctxtloc$.

To capture the use of an identifier in an expression, RepoRep creates the parameterized expressions using the code pattern strings defined in Section III-B (slightly modified by using `$v$` for the RE instance that is targeted) for the immediate ex-

pression the variable is nested in, the expression the immediate expression is nested in, etc. This yields a list of parameterized expressions as *ctxt_exps*. RepoRep scores two instances by finding their longest common parameterized expressions before *ctxt_exps* differs, counting the number of symbols in that expression and multiplying by *sctxtexp*. In the above example, since the longest parameterized expressions between `index` and `index2` is `$p$=$v$;` (the RE under study is `$v$`) from `indexOfMin=index;` and `indexSmallest=index2;`, the expression score is $4 \times$ *sctxtexp*.

RepoRep chooses the top three variable mappings based on their scores. For each of these, it next finds the most appropriate mapping of type identifiers. This is again done heuristically, based on the *type names*, the *def-use instances* where the type names appear, and, most importantly, the *matching variables* declared of each type. The latter takes into account that variables that are mapped to each other should have consistent types. Name matching is done the same way for types as for variables. Similarly, the *def-use instances* where the type names appear are also computed in the same way, except RepoRep discounts the score by multiplying by $stctxtfactor$ since such contexts are not as important based on our experimental observations. The declaration matching score for two type REs is simply the number of variables declared of the two different types that are mapped to each other. Again the final score is the sum of the three matching scores. To extend the name mapping, for each of the three variable mappings, RepoRep finds the best type mapping (i.e., the mapping with the highest score) and augments each of the variable mappings accordingly.

Finally, if there are any method names that need to be mapped, RepoRep finds the best mapping of these names and augments the combined variable and type mapping accordingly. The score for matching two method REs is the sum of scores based on their names and their def-use instances. The name and def-use instance scores are computed as for variables.

The result is 3 complete name matchings for each of the 100 initial candidates. This produces a total of 300 mapped candidates that RepoRep can then use for patch generation.

2) Find the Best Parameter Values: RepoRep uses seven score parameters for creating the name mappings in Section III-C1. They are *sdecltype*, *slname*, *ssname*, *ssloc*, *snloc*, *sctxtsymbol*, *stctxtweight*. To obtain a proper set of values, we trained RepoRep's renaming process over the held-out dataset consisting of 20 buggy programs (Section II). For each buggy program's target method, we manually found a candidate method that can fix the bug. This results in 20 target-candidate pairs. We also manually created an appropriate name mapping for the critical REs as the oracle for each bug-candidate pair. With each set of parameter values that we tried, RepoRep ran the renaming process for each target-candidate pair and checked the best result name mapping against the oracle to decide whether the result name mapping was valid.

We systematically tried all possible value-combinations for the seven parameters under the following constraints to make

```
1 public static void sortAnything(
2 Comparable a[], int numberUsed) {
3   Comparable min;
4   int indexOfMin; int startIndex; int index;
5   for (startIndex = 0;
6     startIndex < numberUsed - 1; startIndex++) {
7     indexOfMin = startIndex;
8     for (index = startIndex + 1;
9          index < numberUsed; index++) {
10      if (a[index].compareTo(
11          a[indexOfMin])) {
12        indexOfMin = index;
13      }
14    }
15    min = a[startIndex];
16    a[startIndex] = a[indexOfMin];
17    a[indexOfMin] = min;
18  }
19 }
```

Fig. 4. Renamed Candidate Example

the experiments affordable. (1) The range is $(0, 1]$ for all parameters except *sctxtsymbol* whose range is $(0, 0.5]$. We choose a smaller range for *sctxtsymbol* because it is multiplied by the number of expression symbols (which is often no less than 4) to compute the matching score of *ctxt_exps*. (2) For each parameter, we chose five values in its range. In specific, we chose $\{0.2, 0.4, 0.6, 0.8, 1.0\}$ for any parameter whose range is $(0, 1]$ and we chose $\{0.1, 0.2, 0.3, 0.4, 0.5\}$ for *sctxtsymbol*. (3) Since two REs with long identical names are often much more likely to match than those with short, single-character names, we define *ssname* to be always greater than *slname*. (4) Since special context locations should be weighted more than normal context locations, we define *ssloc* to be greater than *snloc*. Under these contraints, we have in total 12,500 sets of values. RepoRep was fed with each set of values for the renaming trials. It took about 19 hours to finish. The results show there are 28 out of the 12,500 value-settings that lead to a maximum of 18 valid renamings for the 20 target-candidate pairs. We picked one such value-setting (*sdecltype* $= 0.2$, *slname* $= 1$, *sname* $= 0.2$, *ssloc* $= 0.6$, *snloc* $= 0.2$, *sctxtsymbol* $= 0.5$ and *stctxtweight* $= 0.6$) that we believe can generalize well to be used by RepoRep.

Using this value-setting, a candidate as renamed by RepoRep for our running example is shown in Figure 4, where method `lessThan` is renamed as `compareTo`, type `Sortable` is renamed as `Comparable` and variables `index1`, `numOfItems`, `indexSmallest`, `index2`, `items`, `temp` are renamed as `startIndex`, `numberUsed`, `indexOfMin`, `index`, `a`, `min` respectively.

D. Patch Generation

RepoRep next generates patches for each of the renamed candidates and then evaluates those patches using test cases. Patches are generated based on the differences between the renamed candidate and the target. RepoRep employs ChangeDistiller [29] to match target and candidate at the statement level and identifies any statement differences based on the

matching result. For each such difference, RepoRep makes the corresponding change to the target and yields a patch. The generated patches are then validated using the test cases, and the final patch is chosen based on the results. To actually modify the target's statements, RepoRep uses deletion, replacement and insertion which are also the mutating operations used by GenProg. However, RepoRep's patch generation is essentially different from GenProg's patch generation in three aspects: the modification of the target is performed using the candidate; the modification is deterministic, not probabilistic; and there is no test-case-guided search algorithm such as genetic programming. Without using such algorithm, RepoRep tends to generate less overfitted patches as shown in Section IV-B.

1) Statement Matching: RepoRep employs ChangeDistiller [29] to match the statements of target and candidate using their abstract syntax trees, or ASTs. ChangeDistiller first matches the leaf nodes, and then matches the inner nodes in a bottom-up way. We define the leaf nodes here to be any statement nodes that have no children statements such as the *return* statement. The inner nodes are any statement nodes that have nested statements such as the *if* statement. Two statements can only match if they are compatible. We define two statements are compatible iff their statement types are the same or they are both loop-statements (e.g., *for* v.s. *while* statements). ChangeDistiller matches the leaves based on the bigram string similarity. It matches the inner nodes based on the bigram string similarity of their conditions, if any, (e.g., the *if*-condition) as well as the matchings of their children statements where ChangeDistiller uses the *Dice Coefficient* measure. RepoRep completely follows ChangeDistiller's matching algorithm, its similarity measures and its threshold values to achieve the statement matchings. The matched statements between Figures 2 and 4 can be found in Table II.

2) Modification: RepoRep generates multiple patches based on the statement matching result, validates them against the test cases, and returns any successful one that can pass all the test cases. It performs deletions, replacements and insertions to modify the target based on the matched statements. The modification process is deterministic.

To generate patches, RepoRep performs the three modifications in turn. We used our initial experiments to determine this ordering. It first performs deletions for each of the target statements that are not matched. Note that a target statement s that is not matched may have nested statements that are matched. In that case, RepoRep simply skips s for deletion.

RepoRep next performs the replacement modification. For each statement in the target that has a match in the candidate, RepoRep yields a patch by replacing the statement with its match. In addition, RepoRep yields two more patches by replacing the respective statement condition and the statement body for the following statements: *if*, *do*, *for* and *while* statements. To avoid yielding the same target method, RepoRep performs no modification for any replacement that is identical to its origin.

RepoRep performs the insertions by first looking for statements in the candidate that are not matched. It then attempts to insert each of those statements into some place in the target. Similarly, if a candidate statement s is unmatched but any of its nested statements is matched, RepoRep gives up the insertion for s because the potential occurrence of s in the target could lead to statement redundancy caused by its nested statements. If s is good for insertion, RepoRep attempts to compute an estimate of where s is likely to fit in the target by looking at the statements in the candidate that have matches that come before and after s to be inserted. Let those statements be s_- and s_+ respectively. RepoRep then uses the positions of s_- and s_+ in the target, denoted as s'_- and s'_+, to narrow down the range of possible insertion positions. In specific, if s'_- and s'_+ are from the same block, RepoRep inserts s at each position in between to yield each patch. Otherwise, RepoRep yields patches by inserting s at each position after s'_- in its block and at each position before s'_+ in its block. If neither s_- nor s_+ exists, RepoRep simply gives up inserting s since there is no matching evidence that s is needed.

The total patches are the union of patches yielded after each modification. RepoRep validates each patch against the test cases and reports the successful one passing all of them.

After applying the patch generation techniques to our example target, RepoRep finds a successful patch which is generated by replacing the target for-loop's condition expression `index<=numberUsed` (line 8 of Figure 2) with the corresponding condition expression `index<numberUsed` (line 9 of Figure 4) of the candidate for-loop.

IV. Experiments

To validate our approach, we tested RepoRep using 27 faulty Java programs derived from Fry and Weimer [1], and compared the results to GenProg and Nopol.

A. Experimental Setup

In [1], a study of human's fault localization accuracy, Fry and Weimer collected 45 Java programs from 5 Java textbooks, and manually injected faults into 35 of them. They assumed eighteen different types of bugs and produced a frequency distribution of those types based on their examination of a hundred bug fixes from Mozilla. All inserted faults were simple and traceable to a single line.

Among the 35 faulty programs, we chose 27 and omitted 8: *copyingdemo*, *die*, *heap_d*, *linkedqueue_le*, *stack_s*, *divisiondemofirstversion*, *appletmenudemo* and *coin*. The first five were removed because they contain compiler errors; *divisiondemofirstversion* is inappropriate for automatic testing since it requires keyboard input; *appletmenudemo* requires a graphical environment to be tested; and *coin* involves random expressions which are difficult to test.

For each of the 27 faulty programs, we manually developed both positive and negative test cases. For two of the programs, *lettercollection* and *disjointsetcluster*, we added auxiliary getter methods to access internal states for testing. However, we did not alter any program's target methods or execution.

TABLE II
MATCHED STATEMENTS

Target	**Candidate**
indexOfMin=index;	indexOfMin=index;
if (a[index].compareTo(min)<0){min=a[index];indexOfMin=index;}	if (a[index].compareTo(a[indexOfMin])){indexOfMin=index;}
int indexOfMin=startIndex;	int indexOfMin;
int index;	int index;
for (index=startIndex+1; index<=numberUsed; index++){..}	for (index=startIndex+1; index<numberUsed; index++){..}

We next selected keywords for each program. Because we did not know in advance which method of a program was buggy, we manually chose keywords for each of its possible methods to allow corresponding candidates to be found. This was done using words that came from the faulty program's text and represent the method and the program in some way. For each set of keywords, we cached the searching results to ensure consistency. To preserve the integrity of the repair process, we manually excluded any repository candidates that were from the original program.

For each program, we ran the repair on an Opteron 6282 SE linux machine with 32 cores and 64G memory. For RepoRep and Nopol, we ran the repair once. For GenProg, however, we ran the repair ten times for each program to account for randomness. If any plausible patch was generated that passed all the test cases, the repairing process terminated. It would also terminate with no plausible patch proposed upon completion or when a time limit of 2 hours was reached. After each program's repair, any generated plausible patch was manually examined for correctness. We determine a patch is correct if it is semantically equivalent to the program's non-buggy version in [1].

B. Results

Table IV presents the results. For each program and each tool, the table shows from left to right the average time (in seconds), whether a plausible/correct patch was found (P/C). In addition, the table shows the keywords used by RepoRep. For GenProg, the table shows the number of plausible/correct patches generated (P#/C#) out of ten trials. A repair is plausible if it passed all test cases. We manually determined if each generated repair was semantically correct. A repair is overfitted if it is plausible but not correct as defined in [16], [17].

1) Repair Performance: RepoRep repaired 21 (77.8%) of the 27 faulty programs, and generated correct patches for 19 (70.3%) of them. The patch overfitting rate for RepoRep is only about 9.5% (2/21).

For six of the failing repairs, RepoRep was not able to generate any plausible patches. The failures were due to the dearth of effective candidates being retrieved (*disjointsetcluster*, *employee*, *hanoi_d* and *lettercollection*) and the restrictions in our modification algorithm (*pair*). Currently, RepoRep is not able to handle mutiple fixes. So it fails to repair *quicksort* whose target method contains two bugs. Two repairs were overfitted: the lack of a good candidate and the ease of creating an overfitted patch made it difficult to fix *linkedqueue_c*. *nodepool* deals with intensive node allocations and deallocations. The execution of its target method causes the null-pointer exception. Any generated patch that can avoid this error passes the test case but is unlikely to fix the bug. Though successful, RepoRep took about half an hour to repair *selectionsort* because the faulty chunk was not considered very suspicious. RepoRep wasted its time trying to fix four non-buggy chunks before actually fixing the real faulty one.

2) Comparing to GenProg & Nopol: We next ran experiments comparing GenProg's search-based patching approach and Nopol's [30] synthesis-based approach with RepoRep.

We obtained implementations of GenProg from [31] and Nopol from [30] for Java, and used them to repair the same 27 programs. For each program, we ran GenProg in ten trials (to account for the randomness) and we ran Nopol only in one trial since Nopol is deterministic. The time limit is 2 hours for each trial and for both tools. We used the default settings for the other parameters for both tools.

As shown in table IV, GenProg proposed plausible patches for only seven (25.9%) programs and generated correct patches for four (14.8%) programs. GenProg successfully repaired two programs (*arrayins_la_07* and *dictionaryelement*) by statement deletions and two programs (*linkedqueue_d* and *selectionsort*) by inserting statements which can be found and directly used from the same programs. RepoRep can also repair those four programs with correct patches generated. GenProg has difficulties repairing programs such as *card* and *disjointsetcluster* whose fixes cannot be found as statements within the same program. RepoRep addresses the problem by finding, translating, and using code from the external code repository.

RepoRep is also better in generating correct patches. For the seven programs repaired, GenProg generated overfitted patches for four programs with an overfitting rate 42.9%. The overfitting rate for RepoRep was only 9.5% (C=Yes for 19 out of 21 repaired programs). This is because GenProg's patch search algorithm is test-case-guided whereas RepoRep relies on program comparisons for patch generation.

Unlike RepoRep and GenProg, Nopol is constrained to fixing only condition-related bugs. This results in faster speeds but fewer plausible repairs. Nopol successfully repaired *arraystack_le* by changing its buggy if-condition and *heap_la* and *huffman* both by adding if-conditions that effectively deleted their buggy statements. The results show that Nopol

TABLE III
PROGRAM REPAIR USING DIFFERENT KEYWORDS

Programs	Keywords	Time	P/C
card	card equals	485s	Yes/Yes
	card equals this that	76s	Yes/Yes
	card equals rank	76s	Yes/Yes
	card equals this	68s	Yes/Yes
generalizedselectionsort_s	smallest	1090s	No/No
	generalizedselectionsort indexofsmallest	29s	Yes/Yes
	indexofsmallest min	46s	Yes/Yes
	index min	49s	Yes/Yes
	index min	25s	Yes/Yes
arraystack_d	arraystack pop	43s	Yes/Yes
	array stack pop size	478s	No/No

suffers from generating overfitted conditions with an overfitting rate 50% because it relies on test case execution for constraint formulating and patch synthesizing. RepoRep successfully repaired all the programs that Nopol repaired.

3) Keyword Sensitivity: Since RepoRep requires human-provided keywords as input, it is inevitably sensitive to their selection. To study this we did experiments repairing each program's buggy method using different keywords chosen either from the method's text or to represent the method. In RepoRep, candidates work better if they have REs that are similar to the target and have a similar program structure as the target. At the method level, this is more likely to occur with candidates that perform a similar function to the original method. While simple keywords often work, there are cases where the proper choice of keywords is important. We show the results of three cases in Table III that represent the range of results.

card is a simple class encoding a playing card. The bug is contained in a method "equals" which is used for comparing the equality of two cards. As shown in Table III, although the chosen keywords all lead to successful repairs, "card equals" is not as effective as the other keywords in terms of the repair time. It turns out the useful candidate retrieved by "card equals" is ranked as low as 65 which affects the repairing speed. This means "card equals" does not describe the target well enough.

generalizedselectionsort_s implements the selection sort. We used the buggy method as our illustrative example throughout the paper. It turns out RepoRep cannot really use the keyword "smallest" to fix the bug as shown in Table III. This is because "smallest" is too general to well represent the target method and leads to too many irrelevant methods being retrieved. However, with more specified keywords, RepoRep is able to locate useful candidates to fix the bug.

arraystack_d implements a stack using arrays. The program contains a bug involving the incorrect use of the variable "size" in the "pop" method. It turns out "arraystack pop" is an apppropriate keyword. By breaking "arraystack" into two words and adding the word "size", however, the keyword becomes too constrained and does not lead to correct fixes.

4) Candidate Ranks: To determine if RepoRep was doing extra work by considering 100 candidates and the top three renamed versions (determined by the training data) for each, we recorded the rank of each candidate and its corresponding renamed version used to generate each plausible correct patch. The rank of each candidate is based on its syntactic similarity to the target as described in Section III-B and its renamed version is based on the created name mappings as described in Section III-C. It turns out most correct patches were created by candidates with high ranks: RepoRep is able to obtain 100% repair performance using the top 15 retrieved candidates and can still obtain more than 90% repair performance (in specific, 20 out of 21 plausible patches and 18 out of 19 correct patches) using only the top 5 candidates. Moreover, most correct patches, in specific, 18 out of 21 plausible patches and 17 out of 19 correct patches, were created by the candidates' first renamed versions.

C. Threats to Validity

We have demonstrated the effectiveness of RepoRep for repairing simple bugs in textbook programs. However, the study is limited by the size and complexity of the programs. it is not clear that these results would scale to handle real bugs in large systems: First, our approach is based on the assumption that each program has a single bug within a single method and the bug could be fixed relatively easily. However, the assumption might be too strong for real programs which could contain multiple bugs and a bug could span across multiple methods. Second, methods might not be the suitable form of code chunk for real bugs. This is because methods in large systems are likely to be too unique. Using keywords for code search could also be problematic since finding good keywords for code in real systems can be hard. Third, our heuristics for renaming, code matching and target modification have not been verified on anything other than similar methods. Finally, fault localization for complex systems will be less accurate.

V. RELATED WORK

Our efforts are built on top of significant previous work in a variety of areas.

Searching for Matching Code

RepoRep starts with a keyword search to obtain an initial set of candidates. GitHub [32] provides a keyword interface for searching for code over its large code repositories. Other keyword-based repository search tools include Open Hub [12] and Krugle [33]. While GitHub appears to have the largest repository, the other tools could also work.

Other search tools provide alternatives. Sourcerer [34] allows users to specify both keywords and structural information. CodeGenie [35] accepts test cases for search. S^6 [36] enables users to specify contracts and security constraints in addition to keywords and test cases. Hill et al.'s work [37] explores the linguistic and semantic roles of a query word in the source code. Exemplar [38] takes into account both the program textual description and API calls.

TABLE IV
THE REPAIR RESULTS

	RepoRep			GenProg			Nopol	
Faulty Program	**Time**	**P/C**	**Keywords**	**Time**	**P/C**	**P#/C#**	**Time**	**P/C**
arrayins_la_03	125s	Yes/Yes	swap array	4883s	No/No	0/0	10s	No/No
arrayins_la_07	256s	Yes/Yes	partition left right	16s	Yes/Yes	10/6	14s	Yes/No
arrayiterator	63s	Yes/Yes	arrayiterator hasnext current	94s	No/No	0/0	6s	No/No
arraystack_d	85s	Yes/Yes	arraystack pop	167s	No/No	0/0	8s	No/No
arraystack_le	166s	Yes/Yes	arraystack pop top	210s	No/No	0/0	6s	Yes/Yes
card	79s	Yes/Yes	card equals this that	4045s	No/No	0/0	7s	No/No
dictionaryelement	80s	Yes/Yes	compareto getclass	6s	Yes/Yes	10/10	6s	Yes/No
differentequals	135s	Yes/Yes	equalarrays	7099s	No/No	0/0	9s	No/No
disjointsetcluster	919s	No/No	parent findroot	3837s	Yes/No	1/0	9s	No/No
employee	1024s	No/No	employee compareto	91s	No/No	0/0	6s	No/No
generalizedselectionsort_s	93s	Yes/Yes	index min	950s	No/No	0/0	8s	No/No
hanoi_d	1130s	No/No	hanoi source	3382s	No/No	0/0	10s	No/No
heap_la	131s	Yes/Yes	heap change	7282s	No/No	0/0	6s	Yes/Yes
huffman	247s	Yes/Yes	huffman decode node getleft	128s	No/No	0/0	7s	Yes/Yes
lettercollection	324s	No/No	add letter	1412s	No/No	0/0	7s	No/No
linkedqueue_c	499s	Yes/No	linkedqueue dequeue	18s	Yes/No	10/0	7s	Yes/No
linkedqueue_d	400s	Yes/Yes	linkedqueue add front back	989s	Yes/Yes	7/7	6s	No/No
maxheap_c	961s	Yes/Yes	heap add index	403s	No/No	0/0	13s	No/No
memory	82s	Yes/Yes	swap temp get set	2385s	No/No	0/0	7s	No/No
mergesort_s	291s	Yes/Yes	mergesort array begin end	2234s	No/No	0/0	19s	No/No
nodepool	77s	Yes/No	allocate pool free	10s	Yes/No	10/0	25s	No/No
pair	481s	No/No	pair tostring	54s	No/No	0/0	5s	No/No
pet	119s	Yes/Yes	pet name age weight	4s	No/No	0/0	6s	No/No
quicksort	2150s	No/No	quicksort splitpoint sort	3055s	No/No	0/0	133s	No/No
selectionsort	1914s	Yes/Yes	selectionsort end begin	1142s	Yes/Yes	7/7	69s	No/No
towersofhanoi	219s	Yes/Yes	hanoi tower	1525s	No/No	0/0	17s	No/No
twotypepair	64s	Yes/Yes	equals otherobject	57s	No/No	0/0	5s	No/No

RepoRep prioritizes the initial candidates based on code similarity. Since this can be thought of as finding code clones of the target, existing works of code clone detection [24], [39], [26], [27], [40] are closely related to ours. In particular, we apply the ideas of CCFinder [24] in transforming programs into parameterized tokens for similarity evaluation. Finding code that is syntactically similar (Type-3 clone as defined in [41]) is mostly sufficient for our textbook bugs. However, it seems more realistic to find code that is semantically similar (Type-4 clone) [39], [42] for real bugs. There are other tools finding similar code for plagiarism detection such as MOSS [43] which abstracts programs with k-gram fingerprints and finds code based on the fingerprint overlapping.

Program Matching

Our patch generation starts by matching the target and candidate programs syntactically at the statement level, and patching based on their syntactic differences. It employs ChangeDistiller [29] which uses a tree differencing algorithm that matches programs in a bottom-up way to do so. Yang [44] proposed a similar approach which identifies syntactic differences between two programs by matching their syntax trees. The matching algorithm, however, is based on dynamic programming and is performed in a top-down way. The approach is not suitable for our needs since it is too dependent on top-level similarity whereas we are looking for low-level similarity. Because RepoRep only performs program matching after finding similar code, normalization, and then renaming, its program matching is relatively simple compared to many existing program differencing approaches [45], [46], [47], [48], [49].

Bug Fixing

The core idea of RepoRep is to search for code in the repository that has the potential for bug-fixing (we look for candidate code that is syntactically similar to the target) and then use that candidate code to modify the target and generate patches. Recent repairing techniques by SearchRepair [18] and Code Phage [19] share the same basic idea but achieve the goal in different ways.

SearchRepair searches for code that is semantically similar to the buggy code for its fixing. The semantic similarity is encoded as constraints and finding similar code is done through constraint-solving. Code Phage (CP) directly finds code in the repository having the correct semantics through executing the code with the given inputs. The tools differ from RepoRep essentially in the form of code search. SearchRepair and CP leverage constraint-solving and code execution for semantic code search whereas RepoRep uses paramterized code pattern for syntactic code search. Although semantic code search is in general more precise, code semantics is often hard and expensive to obtain. Constraints used by SearchRepair have limited expressive power and constraint-solving is in general undecidable and expensive. CP's code search by execution is also expensive. Still, it could lose the opportunities of utilizing any programs that cannot execute the inputs but are good for bug fixing. In contrast, RepoRep's syntactic code search is fast and cheap. It has the better potential to deal with any

code repository that is in large-scale and contains a large amount of arbitrary and incomplete code fragments. The tools are also different in terms of candidate translation and target modification: SearchRepair encodes as constraint all possible variable mappings between target and candidate and further checks the constraint's satisfiability. For target modification, it performs the simple candidate code replacement; CP identifies the code from candidate to be inserted in the target and relies on instrumented execution to realize code translation, patch generation and patch validation. In particular, CP is subject to fixing certain types of bugs that need a check which can be found and further inserted into the program; RepoRep uses heuristics to identify identifier's name and context similarities to create name mappings. It performs the target modification at the statement level using the candidate. RepoRep is general and not targeted towards any types of bugs.

Besides using code search, a variety of systems have been developed in recent years looking at other ways to generate patches for a buggy program and validate the patches using a set of test cases, at least one of which exposes the bug. GenProg [7], [8], an early such system, employs genetic operations, mutation and crossover, to create program variants and uses genetic programming to search for patches. AE [15] proposes an adaptive patch search algorithm and leverages program equivalence analysis for reducing the search space. RSRepair [14] applies random search instead of genetic programming along with test case prioritization techniques to generate patches. Their results show that random search can have better performance. A recent study [16] puts into doubts their actual repairing capabilities after finding out "the overwhelming majority of the accepted patches are not correct". PAR [50] inherits the patch search process of genetic programming but uses predefined fix templates to create program variants. However, in [51], Monperrus points out that the fix templates do not address any defect class, and most bugs seem be fixed by only two of the templates. The staged repair technique of SPR [52] applies any of the developed transformation schemas to an identified fault to form a parameterized fixing sketch first. It next performs value search or condition synthesis (if the fix is about conditions) to generate repairs. Prophet [53] uses a trained probabilistic model to speed finding patches that are likely to be correct from its candidate patch space. All of these approaches use fault localization techniques to target more suspicious code.

RepoRep differs from these in that it does not rely on solely creating new code or on using code from the original program as the source for potential changes. Rather, it identifies a buggy chunk and then finds relevant, probably correct code from large code repositories as the source for the patches. The search space of potential patches is thus greatly decreased and RepoRep is more likely to generate correct patches. Since it assumes the buggy code chunk is a method, RepoRep only requires a fault localization technique that can identify the executed methods that are most likely to contain a bug and not a particular line or statement.

Some other techniques use specifications or contracts as oracles and utilize techniques such as constraint-solving and program synthesis to correct bad code. AutoFix-E [54] requires program contracts to reason about the predicates indicating failures. It then synthesizes possible fixes. Gopinath et al.'s approach [55] relies on the provided behavioral specification to yield a counter-example, and then uses SAT solving to fill the parameterized parts of the faulty statements with correct expressions. SemFix [56] uses test cases to build the specification as constraint for any suspicious statement, and then generates the fix conforming to the specification via constraint-solving. RepoRep differs from these techniques in that it does not require or build any form of specifications.

Other approaches look at specific fixing tasks, repair suggestions, and providing feedback. Relifix [57] targets on fixing regression faults. Similar to RepoRep, Relifix's repair first finds candidate and syntactically compare the candidate and the target for bug fixing. The difference is, for Relifix, the candidate comes from the previous version of the buggy program where the program has changed. After fault localization, the candidate can be easily identified. There is also no need to tranform the candidate. RepoRep, however, makes it effort in finding useful candidates that are totally external to the buggy program. It uses heuristics to rename the candidates to be used for target's fixing. Compared to RepoRep, Relifix's modification is more specialized and thus more complex. The modification is non-deterministic. Singh et al. [58] use program synthesis to automatically provide feedbacks for student's incorrect code. Though the goal is similar to ours, they focus on generating feedback rather than patches. For that purpose, their approach requires the solution to the incorrect code and the correction rules. MintHint [59] applies statistical analysis in selecting expressions that are likely to be correct for a faulty statement, and then synthesizes repair hints for it. Demsky et al. [60] detects and then repairs inconsistent data structures. Logozzo et al.'s work [61] suggests repairs based on the violations and errors reported by a program verifier. Works of [62], [63] focus on fixing concurrency bugs.

VI. Conclusion

In this paper, we have shown RepoRep, a new system for automatic program repair. Different from current techniques that use the same faulty program for repair, RepoRep leverages code that potentially contains the corrected behaviors of the bug from external, large code repositories. We have tested RepoRep by successfully repairing textbook buggy programs in a reasonable amount of time. For future work, we aim to extend our system to real bug fixing. Code for RepoRep as well as the examples we used for testing is available upon request.

Acknowledgments

This work is supported by the National Science Foundation grant CCF1130822.

REFERENCES

[1] Z. P. Fry and W. Weimer, "A human study of fault localization accuracy," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Sixth International Symposium on Software Reliability Engineering*, 1995, pp. 143–151.

[3] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering*, 2003, pp. 30–39.

[4] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 342–351.

[5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 15–26.

[7] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, pp. 54–72, 2012.

[8] C. L. 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 *34th International Conference on Software Engineering*, 2012, pp. 3–13.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 57–72.

[10] 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*, 2014, pp. 306–317.

[11] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 147–156.

[12] "https://www.openhub.net."

[13] "https://github.com/about/press."

[14] 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*, 2014, pp. 254–265.

[15] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," in *IEEE/ACM 28th International Conference on Automated Software Engineering*, 2013, pp. 356–366.

[16] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *2015 International Symposium on Software Testing and Analysis*, 2015.

[17] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015, pp. 532–543.

[18] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search," in *IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[19] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *ACM SIGPLAN conference on Programming Language Design and Implementation*, 2015.

[20] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 12, pp. 1236–1256, December 2015.

[21] "http://stackoverflow.com."

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

[23] "http://www.eclemma.org/jacoco."

[24] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, August 2002.

[25] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *13th Working Conference on Reverse Engineering*, 2006, pp. 253–262.

[26] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, March 2006.

[27] 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*, 2007, pp. 96–105.

[28] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

[29] B. Fluri, M. Wursch, P. M., and G. H. C., "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, pp. 725–743, 2007.

[30] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *CSTVA'2014*, 2014.

[31] M. Martinez and M. Monperrus, "ASTOR: Evolutionary automatic software repair for Java," Inria, Tech. Rep., 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01075976

[32] "http://github.com."

[33] "http://www.krugle.com."

[34] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: a search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 681–682.

[35] O. A. L. Lemos, S. K. Bajracharya, J. Ossher, R. S. Morla, P. C. Masiero, P. Baldi, and C. V. Lopes, "CodeGenie: using test-cases to search and reuse source code," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 525–526.

[36] S. P. Reiss, "Semantics-based code search," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 243–253.

[37] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 524–527.

[38] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2011.

[39] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings., International Conference on Software Maintenance*, 1998, pp. 368–377.

[40] M.-W. Lee, J.-W. Roh, S. won Hwang, and S. Kim, "Instant code clone search," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 167–176.

[41] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, May 2009.

[42] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 321–330.

[43] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[44] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, pp. 739–755, 1991.

[45] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 234–245.

[46] M. Kim and D. Notkin, “Discovering and representing systematic code changes,” in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 309–319.
[47] T. Apiwattanapong, A. Orso, and M. J. Harrold, “A differencing algorithm for object-oriented programs,” in *Proceedings of the 19th IEEE international conference on Automated software engineering*, 2004, pp. 2–13.
[48] Z. Xing and E. Stroulia, “UMLDiff: An algorithm for object-oriented design differencing,” in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 54–65.
[49] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen, “iDiff: Interaction-based program differencing tool,” in *26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 572–575.
[50] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *Proceedings of the International Conference on Software Engineering*, 2013, pp. 802–811.
[51] M. Monperrus, “A critical review of ”automatic patch generation learned from human-written patches”: essay on the problem statement and the evaluation of automatic software repair,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.
[52] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 166–178.
[53] ——, “Automatic patch generation by learning correct code,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2016, pp. 298–312.
[54] Y. Wei, Y. Pei, C. A. Furia, S. L. S, S. Buchholz, M. B., and A. Zeller, “Automated fixing of programs with contracts,” in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 61–72.
[55] D. Gopinath, M. Z. Malik, and S. Khurshid, “Specification-based program repair using SAT,” in *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, 2011, pp. 173–188.
[56] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “SemFix: Program repair via semantic analysis,” in *Proceedings of International Conference on Software Engineering*, 2013, pp. 772–781.
[57] S. H. Tan and A. Roychoudhury, “Relifix: Automated repair of software regressions,” in *Proceedings of the 37th International Conference on Software Engineering*, 2015, pp. 471–482.
[58] R. Singh, S. Gulwani, and A. Solar-Lezama, “Automated feedback generation for introductory programming assignments,” in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 15–26.
[59] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, “MintHint: Automated synthesis of repair hints,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 266–276.
[60] B. Demsky and M. Rinard, “Automatic detection and repair of errors in data structures,” in *Proceedings of 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, 2003, pp. 78–95.
[61] F. Logozzo and T. Ball, “Modular and verified automatic program repair,” in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 133–146.
[62] P. Liu and C. Zhang, “Axis: Automatically fixing atomicity violations through solving control constraints,” in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 299–309.
[63] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, “Test-driven repair of data races in structured parallel programs,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 15–25.