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
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-preserving transformations. 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 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$$\$t$\$$\$p$$\$t$”. 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

```java
public static void sortAnything(
        Sortable items[], int numOfItems) {
    Sortable temp;
    int indexSmallest; int index1; int index2;
    for (index1 = 0; index1 < numOfItems - 1; index1++) {
        indexSmallest = index1;
        for (index2 = index1 + 1; index2 < numOfItems; index2++) {
            if (items[index2].lessThan(items[indexSmallest])) {
                indexSmallest = index2;
            }
        }
        temp = items[index1];
        items[index1] = items[indexSmallest];
        items[indexSmallest] = temp;
    }
}
```

Fig. 3. An Example of Candidate Code (Normalized)
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 \( \alpha \)-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` vs. `List`) or incompatible primitive types ¹ (e.g., `int` vs. `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 `ssname` 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 expressions (ctxt_search) and the context expressions (ctxt_exprs). A context corresponds to an instance of the RE instance in the AST where the immediate context is the instance’s parent. `ctxt_locs`

<table>
<thead>
<tr>
<th>Normal</th>
<th>Special</th>
</tr>
</thead>
<tbody>
<tr>
<td>class-field</td>
<td>method-parameter</td>
</tr>
<tr>
<td>class-method</td>
<td>catch-clause-parameter</td>
</tr>
<tr>
<td>method-body</td>
<td>enhanced-for-statement-parameter</td>
</tr>
<tr>
<td>catch-clause-body</td>
<td>enhanced-for-statement-expression</td>
</tr>
<tr>
<td>if-then-branch</td>
<td>loop-condition</td>
</tr>
<tr>
<td>if-else-branch</td>
<td>loop-body</td>
</tr>
<tr>
<td>labeled-body</td>
<td>for-initializer</td>
</tr>
<tr>
<td>switch-body</td>
<td>for-updater</td>
</tr>
<tr>
<td>synchronized-body</td>
<td>labeled-label</td>
</tr>
<tr>
<td>try-body</td>
<td>switch-expression</td>
</tr>
<tr>
<td>finally-body</td>
<td>synchronized-expression</td>
</tr>
</tbody>
</table>

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

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_exprs` is a list of parametered expressions in which an RE appears. For example, `ctxt_exprs` 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.

Now we explain the RepoRep-defined location types and the parameterized 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-condition` 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 `sctxtloc + 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 $\text{ctx}_\text{exp}$. RepoRep scores two instances by
finding their longest common parameterized expressions be-
fore $\text{ctx}_\text{exp}$ differs, counting the number of symbols in that
expression and multiplying by $\text{ctx}_\text{step}$. In the above example,
since the longest parameterized expressions between $\text{index}$
and $\text{index2}$ is $\$p\$=\$v\$; (the RE under study is $\$v\$) from
$\text{index}\text{OFMin}=\text{index}$; and $\text{index}\text{Smallest}=\text{index2}$,
the expression score is $4 \times \text{ctx}_\text{step}$.

RepoRep chooses the top three 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
$\text{ctx}_\text{step}\text{factor}$ 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 accord-
ingly. 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 $\text{idents}_\text{type}$, $\text{name}_\text{shame}$, $\text{ssname}_\text{sp}$, $\text{ssloc}_\text{sl}$,
$\text{ctx}_\text{tsymbol}_\text{symbol}$, $\text{ctx}_\text{tsymbol}_\text{weight}$. 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
the experiments affordable. (1) The range is $(0,1]$ for all
parameters except $\text{ctx}_\text{tsymbol}_\text{symbol}$ whose range is $(0,0.5]$. We
choose a smaller range for $\text{ctx}_\text{tsymbol}_\text{symbol}$ because it is multiplied
by the number of expression symbols (which is often no
less than 4) to compute the matching score of $\text{ctx}_\text{exp}$. (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 $\text{ctx}_\text{tsymbol}$. (3) Since two REs with long identical names
are often much more likely to match than those with short,
single-character names, we define $\text{ssname}$ to be always greater
than $\text{slname}$. (4) Since special context locations should be
weighted more than normal context locations, we define $\text{ssloc}$
to be greater than $\text{slloc}$. Under these constraints, 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
($\text{idents}_\text{type} = 0.2, \text{name}_\text{shame} = 1, \text{ssname} = 0.2, \text{slloc} = 0.6,$
$\text{ssloc} = 0.2, \text{ctx}_\text{tsymbol} = 0.5$ and $\text{ctx}_\text{tsymbol}_\text{weight} = 0.6$) that
we believe can generalize well to be used by RepoRep.

Using this value-setting, a candidate as renamed by Re-
poRep for our running example is shown in Figure 4,
where method $\text{lessThan}$ is renamed as $\text{compareTo}$,
type $\text{Sortable}$ is renamed as $\text{Comparable}$ and var-
iables $\text{index1}$, $\text{numOfItems}$, $\text{index}\text{Smallest}$, $\text{index2}$,
items, $\text{temp}$ are renamed as $\text{startIndex}$, $\text{numberUsed}$,
$\text{index}\text{OFMin}$, $\text{index}$, $\text{a}$, $\text{min}$ 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 re-
named candidate and the target. RepoRep employs ChangeDis-
tillert [29] to match target and candidate at the statement
level and identifies any statement differences based on the

```java
public static void sortAnything(
    Comparable a[], int numberUsed) {
    Comparable min;
    int indexOFMin; int startIndex; int index;
    for (startIndex = 0; 
        startIndex < numberUsed - 1; startIndex++) {
        indexOFMin = startIndex;
        for (index = startIndex + 1; 
            index < numberUsed; index++) {
            if (a[index].compareTo(
                a[indexOFMin])) {
                indexOFMin = index;
            }
        } 
        min = a[startIndex];
        a[startIndex] = a[indexOFMin];
        a[indexOFMin] = min;
    }
}
```

Fig. 4. Renamed Candidate Example
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 vs. 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, linkeddequeue_le, stack_s, divisiondemos Olverversion, appletmenudo and coin. The first five were removed because they contain compiler errors; divisiondemos Olverversion is inappropriate for automatic testing since it requires keyboard input; appletmenudo 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.
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 multiple 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 (arrays_l_a_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_l_e by changing its buggy if-condition and heap_l_a 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

<table>
<thead>
<tr>
<th>Programs</th>
<th>Keywords</th>
<th>Time</th>
<th>P/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>card</td>
<td>card equals</td>
<td>488s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>card equals this that</td>
<td>76s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>card equals rank</td>
<td>76s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>card equals this</td>
<td>68s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>smallest</td>
<td>1090s</td>
<td>No/No</td>
</tr>
<tr>
<td></td>
<td>generalizedselectionsort</td>
<td>29s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>indexofsmallest</td>
<td>46s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>index of smallest</td>
<td>49s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>index min</td>
<td>25s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>array stack pop</td>
<td>43s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td></td>
<td>array stack pop size</td>
<td>478s</td>
<td>No/No</td>
</tr>
</tbody>
</table>

suffers from generating overfitting 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* 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* 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 appropriate 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.
These results indicate that 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 differing 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 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 parameterized 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

<table>
<thead>
<tr>
<th>Faulty Program</th>
<th>Time</th>
<th>P/C</th>
<th>Keywords</th>
<th>Time</th>
<th>P/C</th>
<th>P/#/C#</th>
<th>Time</th>
<th>P/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>arraysins_{la}</td>
<td>125s</td>
<td>Yes</td>
<td>swap array</td>
<td>4883s</td>
<td>No/No</td>
<td>0/0</td>
<td>10s</td>
<td>No/No</td>
</tr>
<tr>
<td>arraysins_{la}</td>
<td>256s</td>
<td>Yes</td>
<td>partition left right</td>
<td>16s</td>
<td>Yes/Yes</td>
<td>10/6</td>
<td>14s</td>
<td>Yes/No</td>
</tr>
<tr>
<td>arrayiterator</td>
<td>63s</td>
<td>Yes</td>
<td>arrayiterator hasnext current</td>
<td>9s</td>
<td>No/No</td>
<td>0/0</td>
<td>6s</td>
<td>No/No</td>
</tr>
<tr>
<td>arraystack_{d}</td>
<td>83s</td>
<td>Yes</td>
<td>arraystack pop</td>
<td>10s</td>
<td>No/No</td>
<td>0/0</td>
<td>8s</td>
<td>No/No</td>
</tr>
<tr>
<td>arraystack_{le}</td>
<td>166s</td>
<td>Yes</td>
<td>arraystack pop top</td>
<td>210s</td>
<td>No/No</td>
<td>0/0</td>
<td>6s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>card</td>
<td>79s</td>
<td>Yes</td>
<td>card equals short that</td>
<td>4045s</td>
<td>No/No</td>
<td>0/0</td>
<td>7s</td>
<td>No/No</td>
</tr>
<tr>
<td>dictionaryelement</td>
<td>80s</td>
<td>Yes</td>
<td>compare to get class</td>
<td>6s</td>
<td>Yes/Yes</td>
<td>10/10</td>
<td>6s</td>
<td>Yes/No</td>
</tr>
<tr>
<td>differentequals</td>
<td>135s</td>
<td>Yes</td>
<td>different equals</td>
<td>7099s</td>
<td>No/No</td>
<td>0/0</td>
<td>9s</td>
<td>No/No</td>
</tr>
<tr>
<td>disjointsetcluster</td>
<td>919s</td>
<td>No</td>
<td>parent findroot</td>
<td>3837s</td>
<td>Yes/Yes</td>
<td>1/0</td>
<td>9s</td>
<td>No/No</td>
</tr>
<tr>
<td>employee</td>
<td>1024s</td>
<td>No</td>
<td>employee compare to</td>
<td>91s</td>
<td>No/No</td>
<td>0/0</td>
<td>6s</td>
<td>No/No</td>
</tr>
<tr>
<td>generalizedselectionsort_{s}</td>
<td>93s</td>
<td>Yes</td>
<td>index min</td>
<td>950s</td>
<td>No/No</td>
<td>0/0</td>
<td>8s</td>
<td>No/No</td>
</tr>
<tr>
<td>hanoi_{d}</td>
<td>1130s</td>
<td>No</td>
<td>hanoi source</td>
<td>3382s</td>
<td>No/No</td>
<td>0/0</td>
<td>10s</td>
<td>No/No</td>
</tr>
<tr>
<td>heap_{la}</td>
<td>131s</td>
<td>Yes</td>
<td>heap change</td>
<td>7282s</td>
<td>No/No</td>
<td>0/0</td>
<td>6s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>huffman</td>
<td>247s</td>
<td>Yes</td>
<td>huffman decode node get left</td>
<td>128s</td>
<td>No/No</td>
<td>0/0</td>
<td>7s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>lettercollection</td>
<td>324s</td>
<td>No</td>
<td>add letter</td>
<td>1412s</td>
<td>No/No</td>
<td>0/0</td>
<td>7s</td>
<td>No/No</td>
</tr>
<tr>
<td>linkedqueue_{c}</td>
<td>499s</td>
<td>Yes</td>
<td>linkedqueue dequeue</td>
<td>18s</td>
<td>Yes/Yes</td>
<td>1/0</td>
<td>7s</td>
<td>Yes/Yes</td>
</tr>
<tr>
<td>linkedqueue_{d}</td>
<td>400s</td>
<td>Yes</td>
<td>linkedqueue add front back</td>
<td>989s</td>
<td>Yes</td>
<td>7/7</td>
<td>6s</td>
<td>No/No</td>
</tr>
<tr>
<td>maxheap_{c}</td>
<td>961s</td>
<td>Yes</td>
<td>heap add index</td>
<td>403s</td>
<td>No/No</td>
<td>0/0</td>
<td>13s</td>
<td>No/No</td>
</tr>
<tr>
<td>memory</td>
<td>82s</td>
<td>Yes</td>
<td>swap temp get set</td>
<td>2385s</td>
<td>No</td>
<td>0/0</td>
<td>7s</td>
<td>No/No</td>
</tr>
<tr>
<td>mergesort_{s}</td>
<td>291s</td>
<td>Yes</td>
<td>mergesort array begin end</td>
<td>2234s</td>
<td>No/No</td>
<td>0/0</td>
<td>19s</td>
<td>No/No</td>
</tr>
<tr>
<td>nodepool</td>
<td>77s</td>
<td>Yes</td>
<td>allocate pool free</td>
<td>10s</td>
<td>Yes/Yes</td>
<td>1/0</td>
<td>12s</td>
<td>No/No</td>
</tr>
<tr>
<td>pair</td>
<td>481s</td>
<td>No/No</td>
<td>pair to string</td>
<td>54s</td>
<td>No/No</td>
<td>0/0</td>
<td>5s</td>
<td>No/No</td>
</tr>
<tr>
<td>pet</td>
<td>119s</td>
<td>Yes</td>
<td>pet name age weight</td>
<td>4s</td>
<td>No/No</td>
<td>0/0</td>
<td>6s</td>
<td>No/No</td>
</tr>
<tr>
<td>quicksort</td>
<td>2130s</td>
<td>No</td>
<td>quicksort splitpoint sort</td>
<td>3055s</td>
<td>No/No</td>
<td>0/0</td>
<td>13s</td>
<td>No/No</td>
</tr>
<tr>
<td>selectionsort</td>
<td>1914s</td>
<td>Yes</td>
<td>selection sort end begin</td>
<td>1142s</td>
<td>Yes/Yes</td>
<td>7/7</td>
<td>69s</td>
<td>No/No</td>
</tr>
<tr>
<td>toserseofhanoi</td>
<td>219s</td>
<td>Yes</td>
<td>hanoi tower</td>
<td>1525s</td>
<td>No/No</td>
<td>0/0</td>
<td>17s</td>
<td>No/No</td>
</tr>
<tr>
<td>twootypepair</td>
<td>64s</td>
<td>Yes</td>
<td>equals other object</td>
<td>57s</td>
<td>No/No</td>
<td>0/0</td>
<td>5s</td>
<td>No/No</td>
</tr>
</tbody>
</table>
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 transform 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


