Leveraging Bug-Related Code for Program Repair

Qi Xin, Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI, USA
{qx5,spr}@cs.brown.edu

Abstract—We present our automatic repair technique, ssFix, which uses syntactic code search to find candidate code that is bug-related and contains the correct fix from both the local project and an external code repository for bug repair. ssFix first identifies suspicious statements in the buggy program through fault localization. For each such statement, ssFix identifies a buggy code chunk which includes the context for the statement. Then ssFix performs code search to obtain candidate code chunks that are both structurally and conceptually similar to the buggy chunk from both the local project and the external code repository. For any of such candidate chunks, ssFix unifies the names used in the candidate chunk with those used in the buggy chunk, establishes the correlation between the two chunks via matching their chunk components as expressions and statements, and generates patches based on the syntactic differences between the chunk components. ssFix finally validates the generated patches with the test suite and reports the first patch that passes all the tests. We demonstrate the effectiveness of ssFix by running ssFix to repair 93 bugs in the Defects4J dataset: ssFix successfully produced correct patches for 22 bugs.

Keywords—bug repair; code search; unification

I. INTRODUCTION

Automatic program repair techniques aim to generate patches for a buggy program, typically at the source level, to correct the program’s faulty behaviors while preserving its normal behaviors. A buggy program is often associated with a test suite with at least one of the test cases exposing the fault(s). The repair techniques should produce patches that pass the test suite and be semantically correct. Since no human efforts are needed, automatic repair techniques are highly desirable: Over the past decade, many repair tools have been developed [1]–[10].

Most of the repair techniques adopt a search-based approach: they define a set of modification rules to generate a space of patches and search in the space for plausible patches that pass the test suite. There are two main challenges faced by such techniques: First, the search space is often huge and it is difficult to search for a correct patch in a huge space. To be worse, the study [11] of two state-of-the-art repair tools SPR [3] and Prophet [4] shows that a huge search space could still be insufficient to cover a correct path and that enlarging the space through using more modification rules would make the repair performance no better. Second, the techniques are prone to produce plausible-but-incorrect patches. According to the study [11], there could be hundreds up to a thousand plausible-but-incorrect patches generated which could simply block the finding of a correct one. Another study [12] also shows that the majority of patches generated by three of the techniques: GenProg [1], AE [2] and RSRepair [13] are incorrect.

Faced with this challenge, recent techniques tend to leverage more fixing information beyond the buggy program and the test suite to produce high-quality patches. One idea is to search for code in a code database that is semantically correct for producing effective patches. Two representatives of the idea are SearchRepair [5] and Code Phage [14]. Although the two techniques demonstrate the feasibility of the idea, finding code that is semantically correct is difficult in general, and it is still unclear how would the techniques deal with repairing real bugs in general: SearchRepair was only shown to work for small C programs and Code Phage is only able to find code that can process the given inputs and it is designed to fix three types of run-time errors whose fixes need checks.

If semantic code search is still limited in finding effective fix code for real bug repair, the natural question would be: Could syntactic code search work? This is the question answered in this paper. Recent studies show that programming languages are repetitive [15] and there exist significant syntactic redundancies in a code database containing 6,000 projects, or 420 million LOC [16]. The syntactic redundancies should be more significant for today’s code repositories which are huge and still rapidly growing (as of 2016, GitHub claims to have 35 million repositories [17]), and it is possible to find code fragments that are syntactically similar to the context of the bug, or are bug-related, in such large code repositories. Among such bug-related code fragments, there may exist code fragments containing the fix recipe. A code fragment containing the fix recipe includes enough information to repair the bug: the fragment may contain the correct forms of expressions or statements that can be used to repair the bug; the fragment may also contain an appropriate code context suggesting a statement in the buggy program should be deleted. Such code fragments containing the fix recipe are also likely to exist in the bug’s local project [18].

We propose our repairing approach, ssFix, which uses syntactic code search to find code fragments that are bug-related but contain the fix recipe from both the bug’s local project and the external code repository and to leverage these code fragments as the fixing candidates to generate patches for the bug. To be useful, a candidate code fragment should have the following properties:

1) It should be syntactically similar (defined in Section III-A) to the context of the bug so that it is likely to
be bug-related and the differences between the candidate and bug can suggest fixes that are likely to be correct.

2) It should contain the fix recipe although it may not be the correct implementation: the candidate may contain more or less functional features than the correct implementation does, it may use different data types or side-effect processing mechanisms, etc.

Although ssFix’s syntactic code search is able to obtain candidates satisfying property (1), it is unable to directly obtain candidates satisfying property (2) without precise semantic validation which can be difficult and expensive. However, our experiments show that it is practical to use the lightweight syntactic code search to find useful candidates for real bug repair and then validate them dynamically.

The main contributions of this paper are two-fold. First, we developed a novel repairing technique, ssFix, which is able to find, via syntactic code search, existing code fragments that are similar to the context of the bug in the code database which consists of the local project of the bug and the external code repository and to leverage these fragments to produce patches that are likely to be correct. Second, we conducted experiments demonstrating the practicality of our technique: ssFix successfully repaired 22 of the 93 simple bugs in the Defects4J bug dataset with correct patches generated (we explain how we selected the bugs in Section IV-A).

II. Overview

Before going into the details, we first give an overview of ssFix. Throughout the paper, we use two bugs as examples showing how ssFix works. The two bugs Lang_33 and Math_33, shown in Figure 2 and Figure 3, are from the Defects4J bug dataset [19]. The buggy code fragment of Lang_33 (at the top of Figure 2) contains a for-loop which intends to get the class type for each array element array[i] and save the type in another array classes. The statement in line 6 is buggy because array[i] could be null. The buggy code fragment of Math_33 (at the top of Figure 3) is part of the implementation for the phase-1 stage of the simplex method. The method compareTo in line 7 misuses the irrelevant threshold maxUlps which is intended to be used as the amount of error for floating-point comparison and is much greater than the correct threshold epsilon that should have been used here for checking the general optimality.

As shown in Figure 1, ssFix goes through four steps: (1) fault localization, (2) code search, (3) patch generation and (4) patch validation to repair a bug. Given a buggy program and the test suite associated with it, ssFix first uses a fault localization technique to identify a statement in the program that is likely to be buggy. ssFix next performs code search to find candidate code fragments that are similar to the context of the bug from the code database. The code database that we use includes the local project of the bug and an external code repository. The code search step is done through three steps: a) buggy chunk identification, b) token extraction and c) candidate retrieval. ssFix next leverages the retrieved candidate code fragments to produce patches for the bug. This is the step patch generation and is also done via three steps: a) candidate translation, b) component matching and c) modification. Finally, ssFix validates the patches against the test suite and reports the first generated patch, if any, that passes the test suite.

A. Fault Localization

ssFix employs the fault localization tool GZoltar [20] to identify statements in the program that are likely to be buggy based on program spectra obtained from test case execution. The result is a list of program statements ranked by their suspiciousness scores from high to low. ssFix scans through the list
in order and works on trying to repair one statement each time. The repair process terminates either when there are no more statements in the list or a time limit is reached (90 minutes for our experiments). For Lang 33 and Math 33, ssFix identifies the statements \texttt{classes[i] = array[i].getClass()} (Figure 2, line 6) and \texttt{columnsToDrop.add(i)} (Figure 3, line 8) as buggy (ranked as No.1 and No.4).

B. Code Search

Given a suspicious statement that is likely to contain a bug, ssFix searches the code database for candidate code fragments that are similar, but not identical, to the context of the bug. A candidate code fragment that has a similar context is likely to be bug-related and the differences between the bug and the candidate may suggest the correct repair. A candidate with a similar context potentially defines a search space of patches that is much reduced so that ssFix is able to find the correct one efficiently. In the ideal case, the only differences between the bug and the candidate are the correct repair and ssFix could thus use the candidate code as the correct patch. ssFix identifies a buggy chunk as \texttt{bchunk} for a buggy statement including the \texttt{local} context of the statement. More details can be found in Section III-A1.

For Lang 33, ssFix identifies the \texttt{bchunk} at the top of Figure 2 which provides the context of the target statement (line 6). The identified \texttt{bchunk} for Math 33 at the top of Figure 3 also defines a reasonable environment for the buggy if statement (line 7).

Next ssFix finds candidate code fragments, \texttt{cchunks}, that are similar to \texttt{bchunk}, from the code database. The similarity is based on both code structure and concept. ssFix extracts both structural and conceptual tokens from \texttt{bchunk} and searches the code database for \texttt{cchunks} having “similar” tokens. More details can be found in Section III-A.

ssFix’s code search finds the candidate chunk (in the middle of Figure 2) for the Lang 33 bug from the code repository and the candidate chunk (at the bottom of Figure 3) from the local class of the \texttt{bchunk} in Math 33. The two candidate chunks are ranked as No.1 and No.4 respectively for bug repair.

C. Patch Generation

ssFix uses a candidate chunk, \texttt{cchunk}, to generate patches for the buggy chunk, \texttt{bchunk}. This is done through three steps: candidate translation, component matching and modification.

a) Candidate Translation: A \texttt{cchunk} retrieved from the code database that is similar to \texttt{bchunk} is likely to use different names for variables, methods and types than those in \texttt{bchunk}. For example, the following two pieces of code uses different names for the loop-index variables, the lists and the length variables that are closely related.

```java
int i = list.size();
for (int i=0; i<l; i++) {...}

int len = lst.size();
for (int index=0; index<len; index++) {...}
```

Without unifying the names in \texttt{cchunk} with those in \texttt{bchunk}, it is difficult for ssFix to find differences that may suggest the correct fix. In Lang 33, the null-checker (line 13) cannot be directly used for the buggy statement (line 6) without changing the array name \texttt{objs} which is simply unrecognizable to the buggy program.

For translation, ssFix builds a name mapping from the names that appear in \texttt{cchunk} to names that appear in \texttt{bchunk} and are accessible in the scope of \texttt{bchunk} (e.g., the global names). There is no need to consider mapping more names that do not appear in \texttt{cchunk} since ssFix will not use them for patch generation. To establish reasonable name mappings, we created heuristics in our prior work [21] for matching variables, types and methods that appear in methods. Variable matching is based on their type compatibility, the original names, and the contexts; type matching is based on the declared variables that are matched, the original names, and the contexts; method matching is based on the original names and the contexts. Using the matching scores, our renaming approach yields the most likely name mappings and the corresponding renamed candidates. More details can be found in [21] where we demonstrated the practicality of our renaming approach via correctly repairing 19 out of 27 programs. ssFix follows the heuristics and scores in the work [21] but uses a variation of the original approach to do translation at the chunk level.

For Lang 33, ssFix maps the variable name \texttt{objs} in \texttt{cchunk} to the name \texttt{array} in \texttt{bchunk} based on their common declared types (this is not shown in the buggy chunk in Figure 2 but is part of the program) and three pairs of def-use instances that ssFix found to be most likely to match:

```java
Class<?>[] classes = new Class[array.length]
Class[] classes = new Class[objs.length]
i < array.length
i < objs.length
```

```java
classes[i] = array[i].getClass()
classes[i] = objs[i].getClass()
```

ssFix computes a score for each matched pair of the def-use instances based on the context patterns and context locations (see [21]). The sum of these scores and the declared type score is the matching score of the two variables. Based on all such matching scores, ssFix yields the top k (we set k as 3 for experiments, see Section IV-A) name mappings and the corresponding renamed candidates. ssFix yields the renamed version of the candidate chunk in Math 33 just as it is as the top version. Note that \texttt{epsilon} was not renamed as \texttt{MaxUIns} because there is an accessible field in the scope of \texttt{bchunk} that has the same name as \texttt{epsilon}.

b) Component Matching: The translated version of \texttt{cchunk}, \texttt{rcchunk}, may represent the correct fix, but more likely, may just contain the fix recipe. For example, in Math 33, although \texttt{rcchunk} (which is also \texttt{cchunk}) is similar to \texttt{bchunk} and contains the correct method call \texttt{Precision.compareTo(entry, 0d, \texttt{epsilon})}, ssFix cannot simply use \texttt{rcchunk} to replace \texttt{bchunk} and get a correct patch. A more feasible approach is to identify the syntactic differences between \texttt{rcchunk} and \texttt{bchunk}. For a bug-related \texttt{rcchunk}, the differences will provide hints on
how to modify \textit{bchunk} and generate patches that are likely to be correct. Matching the components as expressions and statements between \textit{bchunk} and \textit{rchunk} enables ssFix to establish the syntactic correlation between the chunks and thereby to find out their syntactic differences. Based on the identified differences, ssFix is likely to produce a correct patch efficiently.

ssFix uses a modified version of the tree matching algorithm used by ChangeDistiller [22] to do component matching. More details can be found in Section III-B1. ssFix successfully matches the for-loop in \textit{bchunk} (Figure 2, lines 3-7) with the for-loop in \textit{rchunk} (Figure 2, lines 21-25) for Lang\_33, and it maps the method call \texttt{Precision.compareTo(...)} in \textit{bchunk} (Figure 3, line 7) to the method call \texttt{Precision.compareTo(...)} in \textit{rchunk}, or \textit{rchunk} (Figure 3, line 16).

c) \textbf{Modification:} Based on the matched components, ssFix uses three types of modifications: \textit{replacement}, \textit{insertion} and \textit{deletion} in turn to create patches that are likely to be correct. We discuss the three types of modifications with more details in Section III-B2. Using component replacement, ssFix creates the correct patch for Lang\_33 by replacing the buggy for-loop with the correct for-loop in the candidate. It creates the correct patch for Math\_33 by replacing the buggy method call \texttt{Precision.compareTo(entry,0d,\texttt{maxUlps})} (Figure 3, line 7) with the correct method call \texttt{Precision.compareTo(entry,0d,\texttt{epsilon})} (Figure 3, line 16).

\textbf{D. Patch Validation}

ssFix validates the generated patches against the test suite and reports the first patch, if any, that passes the test suite. We call a patch that passes the test suite a \textit{plausible} patch. The correct patches that ssFix generated for Lang\_33 and Math\_33 are both the first plausible patches found during the patch generation process.

\textbf{E. Discussion}

Lang\_33 and Math\_33 are relatively simple bugs, each requiring a simple modification for the generation of a correct patch. It is not difficult for ssFix to repair the bugs through finding fix candidates that are bug-related and using them to make the correct fixing modifications. Without the assistance of such candidates, however, it is not that easy for a general repairing tool to produce correct patches. jGenProg, Nopol and HDRepair, three of the repairing tools that ssFix is compared to in Section IV, failed to generate any correct patches. For jGenProg, the correct patches do not exist in its search space. Although it is possible for HDRepair and Nopol to produce the correct patches for the two bugs, they both failed to find such patches in their huge search spaces. In fact, Nopol did produce a plausible patch that passes the test suite as reported in the experiments conducted by Durieux et al. [23] through replacing the buggy if-condition in Math\_33 with a synthesized condition as shown below.

Additionally, the synthesized condition is unnatural and is incorrect in general. As a general repairing tool, ssFix is not limited to fixing only certain types of bugs. Its current implementation only supports local modifications within a chunk as opposed to non-local modifications, for example, modifications in more than one method.

\section{Methodology}

In this section, we elaborate on two important stages of ssFix: code search and patch generation. Fault localization is currently a research field and is not the focus of this paper. In the fault localization stage, ssFix employs the tool GZoltar [20] to identify suspicious statements. For patch validation, ssFix tests the generated patches against the test suite and reports the first plausible patch found, if any.

\textbf{A. Code Search}

The code search stage of ssFix starts with a suspicious statement \textit{s} identified by fault localization. ssFix generates a chunk \textit{bchunk} for \textit{s} and extracts the structural and conceptual tokens from the text of \textit{bchunk}. ssFix treats the extracted tokens as a vector of terms and uses a tf-idf vector space model to find similar candidate code chunks from the code database. We discuss in detail the generation of a \textit{bchunk} in Section III-A1 and the token extraction in Section III-A2. Section III-A3 explains how we created and indexed the code database. Section III-A4 discusses the similarity metrics used by ssFix for candidate code retrieval.

1) \textit{Generating the Buggy Chunk:} A \textit{bchunk} for \textit{s} with the appropriate code context of \textit{s} included provides information about what \textit{s} intends to do. \textit{bchunk} could thus define a local coding task with the semantics potentially common to a large amount of existing code fragments in the code database. Although it is often desirable to include the context of \textit{s}, using a context that is too large (e.g., a method that implements multiple tasks) generally leads to too many false-positives, since such a chunk is likely to be unique. On the other hand, using a context that is too small yields too many uninteresting code fragments which are often semantically identical to the bug.

To determine an appropriate context for \textit{s}, we did the experiment comparing four types of \textit{bchunks} with different contexts included: (1) \textit{bchunk} that contains only \textit{s}, (2) \textit{bchunk} that contains \textit{s} and the local context of \textit{s}, (3) \textit{bchunk} that contains \textit{s} and the regional context of \textit{s} and (4) \textit{bchunk} that contains the method where \textit{s} is in. A \textit{bchunk} including \textit{s} and its local context is generated using the algorithm shown later in this section where we choose 6 LOC (with no comments or blank lines included) as the maximum size of the chunk (according to the study [16], significant syntactic redundancies were observed for code containing 6-40 tokens, we choose 6 LOC as the maximum size based on preliminary studies). A \textit{bchunk} including \textit{s} and its regional context is generated using
the same algorithm with 12 LOC being the maximum chunk size. We did the experiments on 33 simple bugs in our bug dataset for which we manually found code files containing the fix recipe either from their local projects or from GitHub. For each bug, we ran ssFix to generate the four types of chunks and do code search (see Section III-A2 and Section III-A4) using each type of chunk. We finally obtained the results of the fix code fragments found using each type of chunk for each bug. Our results show the type-2 behunk with local context included for $s$ works the best with the code fragments containing the fix recipe ranked in the top 50 for 17 bugs.

ssFix produces a type-2 behunk for $s$ as follows: Initially, ssFix creates a behunk containing only $s$. It keeps enlarging behunk until the length limit (6 LOC) is reached or behunk is the body block of a declared method. When enlarging behunk, ssFix first gets the statements $sf$ and sl as the first and the last statements in behunk’s body. It next attempts to get the statement $sf'$ that comes before $sf$ in the same block as $sf$ and the statement $sl'$ that comes after $sl$ in the same block as sl. ssFix inserts $sf'$, if it exists, in behunk before $sf$ and inserts $sl'$, if it exists, in behunk after sl. Note that ssFix may insert only one statement, as either $sf'$ or $sl'$, if $sf'$ and $sl'$ do not both exist or inserting both of them will cause the chunk to exceed the length limit. If neither $sf'$ nor $sl'$ exists, ssFix uses the parent statement of $sf$ (or sl) as the behunk unless the size of the parent statement exceeds the length limit.

2) Extracting the chunk tokens: ssFix’s code search attempts to find candidate code fragments, cchunks, that are similar to behunk both structurally and conceptually. ssFix uses structural and conceptual tokens to capture a chunk’s code structure and the concepts contained in a chunk, respectively.

We define a code pattern as a string that characterizes the code structure of a chunk (either behunk or cchunk) by masking the low-level names and literals that appear in the chunk. To obtain the code pattern for a chunk, ssFix tokenizes the chunk’s textual content and then replaces different types of tokens with different symbols: ssFix uses the symbol $/$ for non-JDK-library identifiers, $\$ for non-JDK-library types, $\$ for numbers and $\$ for string literals. ssFix keeps other types of tokens as they are. Note that ssFix does not symbolize the JDK-library tokens and character literals since they are often semantics-indicative. ssFix finally concatenates these tokens to get a code pattern. For example, below is a statement and its code pattern where charAt is a method defined in the JDK library, ‘e’ is a character literal and $\$ for and $\$ for string literals. ssFix keeps other types of tokens as they are. Note that ssFix does not symbolize the JDK-library tokens and character literals since they are often semantics-indicative. ssFix finally concatenates these tokens to get a code pattern. For example, below is a statement and its code pattern where charAt is a method defined in the JDK library, ‘e’ is a character literal and $\$ for and $\$ for string literals. ssFix keeps other types of tokens as they are. Note that ssFix does not symbolize the JDK-library tokens and character literals since they are often semantics-indicative. ssFix finally concatenates these tokens to get a code pattern. For example, below is a statement and its code pattern where charAt is a method defined in the JDK library, ‘e’ is a character literal and $\$ for and $\$ for string literals. ssFix keeps other types of tokens as they are. Note that ssFix does not symbolize the JDK-library tokens and character literals since they are often semantics-indicative.

The code pattern is encoded as a list of k-grams (we set $k=5$ as MOSS [24] does for experiments) to capture the code regularities that significantly exist in programming languages as shown in the study [15]. For example, above code pattern is encoded as a list of four k-grams: \[
\text{charAt}(\$n\$), \text{charAt}(\$n\$)="e", \text{charAt}(\$n\$)="e", \text{charAt}(\$n\$)="e"\]. The extracted k-grams are used as the structural tokens for a chunk.

The conceptual similarity between behunk and cchunk is often reflected as the common usage of words such as “time”, “iterator” or “buffer” appearing in the textual contents of the two chunks. To obtain such concept words for a chunk, ssFix tokenizes the string of the chunk’s code content plus any associated comments to get a sequence of tokens containing only identifiers. All the current tokens are counted as concept words and are added to the word list. For any token that is camel-case or contains underscores, ssFix splits the token into smaller tokens and appends them to the list. ssFix finally changes each token in the list into lower-case and eliminates any tokens whose string lengths are less than 3 or greater than 32 as well as the stop words. For example, the list of concept words for $\text{str}.\text{getChars}(0,\text{strlen},\text{buffer},\text{size})$ is \{“str”, “getChars”, “chars”, “strlen”, “str”, “len”, “buffer” “size”\}. (Note that “get” is a stop word that is eliminated). The concept words extracted as such are used as the conceptual tokens.

3) Indexing the Code Database: ssFix creates an index to facilitate searching over the structural and conceptual tokens. It iterates over Java source files in the code database. For each file, ssFix iterates over the methods in that file. For each method, ssFix extracts the following code fragments as cchunks: a) every sequence of three statements within each basic block and b) every compound statement which contains children statements. cchunks are chosen so as to be likely to contain the fix recipe without suggesting spurious patches while at the same time including enough context to match to behunk. Each cchunk is treated as a document, ssFix extracts the structural tokens and conceptual tokens and employs Apache Lucene [25] to index all the tokens.

4) Searching for Candidate Chunks: During the candidate retrieval stage, ssFix extracts the structural and conceptual tokens from behunk. It uses them as the query tokens to invoke Lucene’s query search. Currently, ssFix uses Lucene’s default tf-idf vector space model. The syntactic similarity score between behunk and cchunk is roughly computed as

\[
\text{overlap}(bts, cts) = \sum_{t \in bts} (tf(t \in cts)) \cdot idf(t)^2
\]

where $t$ is the token (either structural or conceptual), $bts$ is the token list of behunk, $cts$ is the token list of cchunk. overlap(bts, cts) is a score factor based on how many tokens in $bts$ are found in $cts$, $tf(t \in cts)$ is the token frequency for the token $t$ which appears in $cts$, $idf(t)$ is the inverse document frequency for the token $t$. The complete score computation involves some other normalization and boosting factors that are used in Lucene’s default model.

B. Patch Generation

In patch generation stage, ssFix leverages each candidate chunk cchunk to produce patches for behunk. This is done via three steps: candidate translation, component matching and modification. Candidate translation ensures that the names in cchunk are compatible with those in behunk. Details about
candidate translation (or candidate renaming) can be found in our earlier work [21]. We next discuss the latter two steps with more details in turn.

1) Component Matching: Matching the chunk components enables ssFix to establish the correlation between $bchunk$ and $cchunk$ at the expression/statement level and thereby to identify the syntactic differences between the chunks to suggest the likely places for bug repair.

ssFix extends ChangeDistiller’s tree matching algorithm (Fig. 9 in [22]) to match components between the two chunks. The matching is based on the abstract syntax trees of the two chunks where ssFix treats the compound statements (statements having children statements) as the inner nodes and the atomic statements (statements having no children statements) plus the non-name and non-literal expressions as the leaf nodes. ssFix matches the leaf nodes first, and then matches the inner nodes in a bottom-up way. Two nodes can match iff they are compatible. Two leaf nodes are compatible if their component types are equal (e.g., both are return statements). For two leaf nodes having the same component type, we created specific compatibility rules based on the component type, see Table I. We define two inner nodes to be compatible if their statements types are equal or they are both loop-statements (for or while or do statements). ssFix matches leaves based on the bigram string similarity. It matches the inner-nodes based on the matchings of their children expressions and statements where the Dice Coefficient [26] is used as the similarity measure. Note that matching two inner nodes that are both if-statements or loop statements in the original algorithm also depends on the string similarity of their conditions as if-conditions or loop conditions. ssFix does not consider the similarity of two conditions as a factor to determine the match of the statements: a bug could make one condition dissimilar to the other. In such case, ssFix still allows the two statements to match as long as they have similar children according to the Dice Coefficient measure so that the buggy condition has a chance of being repaired.

The original tree matching algorithm of ChangeDistiller was designed to match nodes that are highly similar for evolutionary analysis. In the context of bug repair, we decrease the matching thresholds to allow components that are related but not highly similar to match. Currently ssFix uses 0.2 as the threshold of the bigram similarity for matching leaf nodes and 0.4 as the threshold of the Dice Coefficient similarity for matching inner nodes.

2) Modification: ssFix performs replacement, insertion and deletion in turn to produce patches for $bchunk$ using a translated version of $cchunk$, or $rchunk$. It may produce more patches using more translated versions of $cchunk$ (we use the top three translated versions of $cchunk$ for experiments). ssFix filters away patches that are identical to each other. For efficiency, ssFix only selects the first $k$ (we set $k$ as 50 for experiments, see Section IV-A) patches generated, if any, to be validated against the test suite. Since a buggy chunk is often small as described in Section III-A, the number of patches generated by ssFix using a similar candidate chunk is generally small. If the candidate chunk is not similar, ssFix only selects the first 50 generated patches for validation. We next discuss the three types of modifications in turn.

a) Replacement: Given a matched component pair ($bcpt$, $ccpt$), ssFix first produces a patch by replacing $bcpt$ with $ccpt$ if the two components are not identical. It may produce additional patches by replacing the sub-components of $bcpt$ with those of $ccpt$ based on the types of $bcpt$ and $ccpt$. Table II shows the replacement rules for different component types. Using each rule, ssFix produces a new patch for $bcpt$ if the sub-components of $bcpt$ to be changed are not identical to the corresponding sub-components of $ccpt$. For example, given $bcpt$ and $ccpt$ are both for-statements, ssFix produces a patch replacing the loop-condition of $bcpt$ with the loop-condition of $ccpt$ if the two conditions are not identical using Rule 2 for for-statements. ssFix may produce up to 4 patches using each of the 4 replacement rules for for-statements. Plus the first patch generated by replacing $bcpt$ with $ccpt$, ssFix could produce up to 5 patches for two for-statements. When $bcpt$ and $ccpt$ are both method calls, ssFix may produce more multiple patches, following Rule 3 for method calls, by replacing each individual argument of $bcpt$ with the corresponding argument of $ccpt$ that

<table>
<thead>
<tr>
<th>Component</th>
<th>Rule</th>
</tr>
</thead>
</table>
| If Statements | 1. Replace condition  
| | 2. Replace then-branch  
| | 3. Replace else-branch  
| | 4. Combine conditions with &&  
| | 5. Combine conditions with || |
| For Statements | 1. Replace initializers  
| | 2. Replace condition  
| | 3. Replace updaters  
| | 4. Replace body |
| Loop Statements (not both as for-loops) | 1. Replace condition  
| | 2. Replace body |
| Switch Statements | 1. Replace expression  
| | 2. Replace body |
| Try Statements | 1. Replace try-body  
| | 2. Replace catch-clauses  
| | 3. Replace finally-body |
| Assignments/Infix Expressions | 1. Replace left-hand side  
| | 2. Replace operator  
| | 3. Replace right-hand side |
| Method Calls | 1. Replace invoking expression  
| | 2. Replace method name  
| | 3. Replace arguments (*) |
| Prefix/Postfix Expressions | 1. Replace operator  
| | 2. Replace operand |

TABLE I
COMPATIBILITY RULES FOR COMPONENTS AS LEAF NODES

<table>
<thead>
<tr>
<th>Component Type</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayAccess</td>
<td>Compatible array types</td>
</tr>
<tr>
<td>ArrayCreation</td>
<td>Compatible array types</td>
</tr>
<tr>
<td>ClassInstanceCreation</td>
<td>Compatible class types</td>
</tr>
<tr>
<td>InfixExpression</td>
<td>Same operator</td>
</tr>
<tr>
<td>PostfixExpression</td>
<td>Same operator</td>
</tr>
<tr>
<td>PrefixExpression</td>
<td>Same operator</td>
</tr>
<tr>
<td>MethodInvocation</td>
<td>Same method name</td>
</tr>
<tr>
<td>Assignment</td>
<td>Same assignment operator</td>
</tr>
</tbody>
</table>

TABLE II
REPLACEMENT RULES
is of the same argument index. This only happens when the two method calls have the same method name and the same number of arguments. ssFix is able to produce multiple fixes using one replacement. For example, it may follow Rule 2 for loop statements to replace a loop body with another which could make changes to several statements within the body.

b) Insertion & Deletion: For any \texttt{ccpt} as a statement \texttt{s} in \texttt{rechunk} that is unmatched to any statement in \texttt{behunk}, ssFix attempts to insert \texttt{s} into \texttt{behunk} at estimated positions to yield patches. If \texttt{s} is a statement that has nested children statements and at least one of its children statements is matched, ssFix ignores the insertion for \texttt{s} because the potential occurrence of \texttt{s} in \texttt{behunk} could lead to statement redundancy caused by its children statements. If \texttt{s} is good for insertion, ssFix attempts to compute an estimate of where \texttt{s} is likely to fit in \texttt{behunk} by looking at the statements in \texttt{rechunk} that have matches that come before and after \texttt{s} to be inserted. Let those statements be \texttt{s}_- and \texttt{s}_+ respectively, ssFix then uses the positions of \texttt{s}_- and \texttt{s}_+ in \texttt{behunk}, denoted as \texttt{s'}_- and \texttt{s'}_+, to narrow down the range of possible insertion positions. Specifically, if \texttt{s'}_- and \texttt{s'}_+ are from the same block, ssFix inserts \texttt{s} at each position in between to yield each patch. Otherwise, ssFix yields patches by inserting \texttt{s} at each position after \texttt{s'}_- in its block and at each position before \texttt{s'}_+ in its block. If neither \texttt{s}_- nor \texttt{s}_+ exists, ssFix ignores the insertion for \texttt{s} since there is no matching evidence that \texttt{s} is needed.

For deletion, ssFix deletes any component \texttt{bcpt} as a statement in \texttt{behunk} that is unmatched. Similar to insertion, if the unmatched statement has matched children statements, ssFix ignores its deletion.

IV. EXPERIMENTS

We ran ssFix on a bug dataset consisting of 93 real Java bugs from the Defects4J bug dataset [19] which contains 357 real bugs from 5 open source programs: JFreeChart, Closure Compiler, Commons Lang, Commons Math and Joda-Time. Our dataset is a subset of the Defects4J dataset and contains bugs for which the fixes are relatively simple (see Section IV-A). The results show that ssFix repaired 16 bugs that are not from the Closure-Compiler program (non-Closure) automatically with correct patches generated. With manual fault localization, ssFix produced correct patches for 7 bugs from the Closure-Compiler program (Closure). The average time for generating a patch is less than 20 minutes. Overall, ssFix has a better performance than three existing repair tools it is compared to: jGenProg (the Java version of GenProg [1]), HDRRepair (the history-driven program repair technique [10]) and Nopol [8].

A. Experimental Setup

Each bug in the Defects4J dataset is associated with a human-written correct patch. We looked at each such patch to determine how hard a fix can be and selected 93 of the bugs for which the fixes are relatively simple. A fix is simple if it modifies only one location of the program and no more than 10 LOC. Table III shows the information of the bug dataset where the column “Bug Projects” shows the five projects contained in the Defects4J bug dataset, “Abbrv” shows the abbreviations of the projects, “#Bugs” shows the number of bugs from each project, “#Tests” shows the number of test cases in the test suite associated with each project, and “#Bugs Selected” shows the number of bugs in our bug dataset selected from each project. We use the Merobase dataset [27] (about 2.5 million Java source files, 182 million LOC) as the code repository in the code database and indexed all the Java source files. We also indexed the early versions of the five programs in the Defects4J dataset (C8, C114, L6, M33 and T4) as the local projects. For each bug to be repaired by ssFix, we avoided using its fixed version that might be found in our database to produce patches. For code search, ssFix selects the top 100 candidate chunks for repairing a buggy chunk. For patch generation, ssFix generates the top three renamed versions for each candidate chunk and uses each in turn to produce patches. For efficiency, ssFix only selects the first 50 patches generated, if any, using a candidate chunk to be validated against the test suite, as explained in Section III-B2. We experimentally found that GZoltar failed to identify suspicious statements for the Closure bugs as was shown and explained in [23]. To still demonstrate the potential of ssFix for repairing the Closure bugs, we manually identified one suspicious statement for each of the Closure bugs and ran the experiments. We ran each repair using ssFix in a time limit of 90 minutes on a machine with four AMD Phenom(tm) II processors and 16G memory.

B. Results

1) ssFix’s Performance: Table IV shows the repair results for all bugs for which ssFix found plausible patches. (Bugs that are correctly repaired are marked with *.) A patch is plausible if it passes the test suite. ssFix finally produced plausible patches for 37 bugs. We analyzed each of the generated plausible patches and found 22 of them are correct. We considered a patch to be correct if it removes the bug exposed by the negative test cases without introducing new

<table>
<thead>
<tr>
<th><strong>TABLE III</strong></th>
<th><strong>THE BUG DATASET</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bug Projects</strong></td>
<td><strong>Abbrv</strong></td>
</tr>
<tr>
<td>JFreeChart</td>
<td>C</td>
</tr>
<tr>
<td>Closure Compiler</td>
<td>Cl</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>L</td>
</tr>
<tr>
<td>Commons Math</td>
<td>M</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>T</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>TABLE IV</strong></th>
<th>*<em>BUGS WITH PLAUSIBLE PATCHES GENERATED BY ssFIX (BUGS WITH CORRECT PATCHES GENERATED ARE MARKED WITH <em>)</em></em></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bug Projects</strong></td>
<td><strong>Bug ID</strong></td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1, 8, 17, 20*, 24*</td>
</tr>
<tr>
<td>Closure Compiler</td>
<td>7, 14*, 31*, 40*, 62, 70*, 73*, 86*, 92, 126*</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>6*, 21*, 33*, 39, 43*, 44, 45, 58, 59*</td>
</tr>
<tr>
<td>Commons Math</td>
<td>1, 8, 17, 20*, 24*, 57, 58, 70*, 73*, 92, 126*</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>-</td>
</tr>
</tbody>
</table>
HDRepair on the Closure bugs. For each Closure bug, we compared ssFix to three repair tools: GenProg, HDRepair and Nopol. GenProg [1] is a representative of the search-based repair techniques using a genetic algorithm for patch generation. The history driven repair technique [10], which we call HDRepair, is a search-based technique which leverages common bug-fix patterns mined from 3,000 bug fixing instances to search for patches that are more likely to be correct. Nopol [28] is an approach that uses value search and program synthesis to repair bugs related to any if conditions. GenProg [1] is a representative of the search-based techniques using a genetic algorithm for patch generation. The history driven repair technique [10], which we call HDRepair, is a search-based technique which leverages common bug-fix patterns mined from 3,000 bug fixing instances to search for patches that are more likely to be correct. Nopol [28] is an approach that uses value search and program synthesis to repair bugs related to any if conditions. GenProg [1] is a representative of the search-based techniques using a genetic algorithm for patch generation. The history driven repair technique [10], which we call HDRepair, is a search-based technique which leverages common bug-fix patterns mined from 3,000 bug fixing instances to search for patches that are more likely to be correct. Nopol [28] is an approach that uses value search and program synthesis to repair bugs related to any if conditions. GenProg [1] is a representative of the search-based techniques using a genetic algorithm for patch generation.

We used the same 90-minute time limit for each tool. Since jGenProg and HDRepair use stochastic algorithms, we ran both tools in three trials for each bug. Since Nopol is deterministic, we only ran Nopol once for each bug. We used the same 90-minute time limit for each tool.

Our results are shown in Table V and in Table VI where the columns “#Plausible” and “#Correct” show the numbers of plausible and correct patches created by each tool. “Time” shows the minimum, median, maximum and average time, in minutes, for each tool to produce a plausible patch. (We considered jGenProg or HDRepair to have a plausible/correct patch generated for a bug, as long as the tool did so in at least one of the three trials. Note that although HDRepair is able to produce multiple plausible patches for a bug, we only considered the first generated plausible patch, if any, for comparison.) As the results show, ssFix is able to produce correct patches for more bugs than the other tools with the average time for generating a patch being comparable. ssFix also has the highest rates for yielding a non-overfitting patch, calculated as #Correct/#Plausible, which are 55.6% and 70% for non-Closure and Closure bugs respectively.

We note that given our 90-minute time limit, the results may not show the full potentials of the tools. Since Durieux et al. also conducted experiments using jGenProg and Nopol to repair the Defect4J bugs with the generated patches shown in [32], we also considered the generated patches that are correct in their experiment results (for the same bugs we ran on) for a comparative patch analysis. (As reported in [23], Nopol produced 4 correct patches, but we found the patches generated for L44 and L58 are not actually correct in general although they passed the test suite.)

Figure 4 shows all bugs the four tools correctly repaired. There are eight bugs that can be either correctly repaired by ssFix or by one of the three tools: jGenProg, HDRepair and Nopol. For two of them, M50 and M70, the correct patches were found by three of the four tools. M50 is a bug that can be correctly repaired via statement deletion. ssFix and jGenProg simply deletes the buggy statement to yield the correct patch. Nopol adds an if-statement for the buggy statement with the if-condition always evaluated to false. M70 is a bug for which the fix ingredient can be found within the local program. The buggy statement is a method call as solve(min,max) and the fix statement is a similar method call with same method name but one more argument as solve(f,min,max). The fix statement was successfully found and leveraged by ssFix, jGenProg and HDRepair to produce the correct patch. Since the bug is not related to any if-condition, Nopol simply failed the repair. M53 and L43 are two bugs for which statement insertions are needed where the statements to be inserted can be found in their local programs. Using genetic algorithms, jGenProg and HDRepair each repaired one of the bugs, whereas ssFix successfully repaired both via finding code that has similar contexts. Cl70, Cl73 and M79 are three bugs repaired by HDRepair through simple modifications as changing the method call argument, changing the infix expression and inserting the type-cast with

<table>
<thead>
<tr>
<th>Tool</th>
<th>#Plausible</th>
<th>#Correct</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssFix</td>
<td>10</td>
<td>7</td>
<td>6.3</td>
</tr>
<tr>
<td>HDRepair</td>
<td>11</td>
<td>5</td>
<td>5.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th>#Plausible</th>
<th>#Correct</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssFix</td>
<td>27</td>
<td>15</td>
<td>1.4</td>
</tr>
<tr>
<td>jGenProg</td>
<td>6</td>
<td>3</td>
<td>2.9</td>
</tr>
<tr>
<td>HDRepair</td>
<td>7</td>
<td>3</td>
<td>3.3</td>
</tr>
<tr>
<td>Nopol</td>
<td>1</td>
<td>0</td>
<td>73.3</td>
</tr>
</tbody>
</table>
the support of its corresponding operators. The correct patches were practically found using its fix-pattern-guided search algorithm. For the same bugs, ssFix also efficiently found the patches by leveraging context-similar fix code from their local programs. For Cl126, ssFix and HDRepair both delete the buggy statement and produce a correct patch. 

There are in total four bugs, M5, L55, Cl10 and Cl51, repaired by one of the tools: jGenProg, HDRepair or Nopol, but not by ssFix. M5 is the only bug for which jGenProg found a correct patch while ssFix did not. The fix statement does exist in the bug’s local program and jGenProg successfully used it to yield the correct patch. ssFix generated a code chunk (as shown below) for the buggy statement return NaN.

```java
if (isNaN) { return NaN; }
if (real == 0.0 && imaginary == 0.0) { return NaN; } //Should be changed to "return INF;"
```

However, the chunk does not indeed help ssFix find the fix statement return INF in the local program simply because the code containing the fix statement (as shown below) is not similar to the buggy chunk, and ssFix did not find other program fragments that contain the fix statement or any statement that is semantically identical to the fix statement.

```java
if (Double.isInfinite(real)) { return INF; }
if (Double.isInfinite(imaginary) ||
    Double.isInfinite(factor)) { return INF; }
```

For L55, Nopol created an if-statement with its synthesized if-condition being the valid precondition for another statement. ssFix failed to find any code fragments that provide the if-statement with its condition as the correct precondition for the buggy statement. For Cl10 and Cl51, the correct expressions and statements do not actually exist in the buggy program, ssFix thus failed to leverage any code from their local programs to produce correct patches. By looking at the code search results, we also found there are no related candidate chunks retrieved from any non-local programs in the database that can be actually used for repair. HDRepair, however, produces a slightly mutated method call for Cl10 and adds a condition expression for an if statement for Cl51. Given the suspicious statements, HDRepair successfully produced the mutated statements with correct patches yielded using its search algorithm.

Although ssFix failed four bugs that are correctly repaired by at least one of the other three tools, ssFix correctly repaired 14 bugs that none of the three tools succeed as shown in Figure 4. jGenProg failed the bugs because either the correct statements do not exist in the original program (C24, L6, L21, L33, L59, M33 and M94) or they exist but jGenProg failed to use them to yield the correct patches given the large search space (C20, L43, M57, M58 and M79). HDRepair uses more mutation operators than GenProg does and this makes more patches covered by its search space. In fact, we found most of the correct patches (all the correct patches except L21) do exist in the search space of HDRepair. However, HDRepair failed to repair the bugs since using more operators makes the search space much larger and the search algorithm guided by mined fix-patterns is limited in finding a correct patch: HDRepair failed 9 of the bugs for which the fixes are either simple variable replacement or type casting (C24, M33, M57, L6, L59, Cl14, Cl31, Cl40 and Cl86). (Note that our results are a little different from the results shown in the paper [10] where the repair experiments are based on the assumption that a buggy method is known in advance.) There are also bugs for which the fixes require more than one mutation: fixing C20 needs the replacements of two method arguments and fixing L33 needs a null-checker. In those cases, HDRepair also failed. Among the non-Closure bugs that ssFix correctly repaired but Nopol failed, 7 of them (C20, C24, L6, L21, L43, L59 and M79) are not within the fixing scope of Nopol: they are not bugs related to any if-conditions. For the other bugs, Nopol either failed to produce any correct if-conditions (L33 and M94) or it produced the synthesized if-conditions that are unnatural and incorrect (M33, M57, M58).

3) Threats to Validity: Our approach relies on a candidate chunk retrieved from the code database to produce patches for a bug. Therefore, the repair performance is dependent on the size of the code database especially for bugs for which fixing candidate chunks cannot be found from their local programs. Our approach is based on the assumption that given a buggy chunk, there exist bug-related fix code in the code database, so it is not reasonable to use a code database that is too small. However, although our database is not small, it is not comparable in its size to today’s large code repositories, e.g., GitHub: the fixing candidate chunks that do not exist in our database are more likely to exist in GitHub. We think ssFix could work better with larger code repository and plan to validate this as future work.

To determine whether a plausible patch which passes the test suite is actually correct, we manually examined the patch following the human-written patch associated with the bug to see whether the plausible patch actually removes the bug exposed in the failed test cases and does not introduce new bugs. We think the generated patches for the 22 bugs are correct in general, but it is still possible there are extreme cases exposing errors that exist in those patches that we are not aware of.

Our results were based on a set of relatively simple bugs that we manually selected from the Defects4J bug dataset. The results may not generalize to all the Defects4J bugs. We think none of the repair tools is currently good at repairing more complex bugs, but we did not actually validate this through experiments. Our experiments for repairing the Closure bugs were based on manual fault localization, so the results for Closure bugs may not reflect the tools’ automatic repairing abilities.

The original implementation of GenProg was for C, so it is possible that jGenProg, as the Java implementation of GenProg, does not match the original implementation details. For comparative experiments, we ran jGenProg and HDRepair which use stochastic algorithms in three trials, but it may still be insufficient to show the full potentials of these tools.
V. RELATED WORK

Our work is related to many works in the areas of code search, program repair, and code transfer.

**Code Search** ssFix’s code search is related to a variety of techniques for code clone detection [33]–[37] and plagiarism detection [24], [38], [39]. The main difference between ssFix and such techniques is that ssFix is not targeted on finding candidate chunks that are highly similar to the buggy chunk as clones or plagiarized code fragments. Rather, it attempts to find chunks that are related based on the matched tokens extracted from the chunks. It is thus very related to MOSS [24] which also uses k-grams for finding similar code, but different from MOSS, ssFix uses all the k-gram tokens within the chunk and it also uses concept tokens for code search. Many techniques [40]–[44] leverage the API usage, such as the API calls, in the source program to detect and recommend similar programs. Similar to such techniques, ssFix leverages the API usage for code search through encoding the API usage patterns in the structural tokens extracted from the chunk. ssFix is related to existing techniques [45], [46] that aim to find similar programs with shared high-level concepts. Compared to these techniques, ssFix’s conceptual search is simpler: it finds similar code fragments that share the extracted concept words, but is more IR-oriented and can be easily combined with the structural search.

**Program Repair** Within the past decade, a variety of automatic program repair techniques have been developed. GenProg [1], [47] employs genetic operations to create program variants and uses genetic programming to search for patches. AE [2] proposes an adaptive patch search algorithm and leverages program equivalence analysis for reducing the search space. RSRepair [13] applies random search instead of genetic programming along with test case prioritization techniques to generate patches. The study [12] puts into doubts their actual repairing capabilities after finding out “the overwhelming majority of the accepted patches are not correct”. PAR [48] inherits the patch search process of genetic programming but uses predefined fix templates to create program variants. However, in [49], 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. SemFix [6], DirectFix [7] and Angelix [9] use constraint solving and synthesis techniques to generate patches. Nopol [8] uses value search and program synthesis to repair bugs that are related to the if-conditions. The staged repair technique of SPR [3] applies transformation schemas to an identified fault to form a parameterized fixing sketch first and then performs value search or condition synthesis to generate repairs. Recent repair techniques attempt to leverage fixing information beyond the buggy program and the test cases to produce patches with possibly high qualities. Prophet [4] uses a trained probabilistic model to speed finding patches that are likely to be correct from its candidate patch space. The history driven repair [10] uses GenProg’s patch search framework but leverages the bug-fixing statistics based on the mined bug-fixing instances to improve the repair performance. Genesis [50] leverages existing human patches to infer the likely program transforms and the search space of patches. Tan et al. [51] propose to use the anti-patterns as the forbidden transformations for search-based repair techniques to improve the quality of their generated patches.

Compared to such techniques, ssFix uses a different method to achieve the goal: it finds code in the code database that is likely to contain the fix ingredient and leverages such code to produce patches. Similar to ssFix, SearchRepair [5] is a repairing technique that is also based on code search. The main difference between SearchRepair and ssFix lies in the form of code search: SearchRepair attempts to find code that is semantically similar to the buggy code. The semantic similarity is encoded as constraints derived from symbolic execution and the code search is actually done via constraint-solving. However, scalability seems to be the main challenge for SearchRepair: SearchRepair was only shown to work for small C programs. Currently, it is still unclear how SearchRepair could generally work for real bugs. Compared to SearchRepair, ssFix which uses the lightweight, syntactic code search has the better potential for repairing real-bugs.

ssFix is also related to many repairing techniques using formal specifications [52], focusing on specific fixing tasks [53], [54] and providing suggestions and feedbacks [55], [56].

**Code Transfer** Our work is related to several techniques of code transfer [14], [57], [58] which aim to find desirable code pieces from a code database and create a hybrid of the original program and the found code pieces to implement the desired behavior. ssFix’s repair can be thought of as transferring the correct expressions/statements from the candidate chunk to the buggy chunk. Code Phage [14], or CP, is also able to transfer the correct code to the target program for error elimination. However, CP is different from ssFix in both code search and patch generation. First, CP can only find code that can process the test inputs: it needs to execute the inputs to identify the correct code to be used for error elimination. Compared to CP, ssFix’s syntactic code search is more flexible and is able to find similar code fragments at the source level without executing the programs. Second, CP only targets on three types of errors whose fixes need checks whereas ssFix is able to repair bugs in general. Third, CP works at the binary level while ssFix works at the source level.

VI. CONCLUSION

We presented ssFix which performs syntactic code search to find candidate code fragments that are syntactically similar to the context of the bug from both the bug’s local project and the external code repository. Without performing any expensive semantic analysis, ssFix assumes such bug-related code fragments to be likely to contain the fix recipe for bug repair and it leverages such candidate fragments to produce patches via candidate translation, component matching and modification. Our experiments show that ssFix is more effective than the current repair tools jGenProg, HDRepair and Nopol to produce correct patches for real bugs.