Leveraging Syntax-Related Code for Automated 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 automated program repair technique ssFix which leverages existing code (from a code database) that is syntax-related to the context of a bug to produce patches for its repair. Given a faulty program and a fault-exposing test suite, ssFix does fault localization to identify suspicious statements that are likely to be faulty. For each such statement, ssFix identifies a code chunk (or target chunk) including the statement and its local context. ssFix works on the target chunk to produce patches. To do so, it first performs syntactic code search to find candidate code chunks that are syntax-related, i.e., structurally similar and conceptually related, to the target chunk from a code database (or codebase) consisting of the local faulty program and an external code repository. ssFix assumes the correct fix to be contained in the candidate chunks, and it leverages each candidate chunk to produce patches for the target chunk. To do so, ssFix translates the candidate chunk by unifying the names used in the candidate chunk with those in the target chunk; matches the chunk components (expressions and statements) between the translated candidate chunk and the target chunk; and produces patches for the target chunk based on the syntactic differences that exist between the matched components and in the unmatched components. ssFix finally validates the patched programs generated against the test suite and reports the first one that passes the test suite.**

We evaluated ssFix on 357 bugs in the Defects4J bug dataset. Our results show that ssFix successfully repaired 20 bugs with valid patches generated and that it outperformed five other repair techniques for Java.

***Index Terms*—Automated program repair; code search; code transfer**

I. Introduction

A typical automated program repair technique accepts as input a faulty program and a fault-exposing test suite. As output, it produces patched programs that pass the test suite. A significant fraction of current repair techniques adopt a search-based approach [1]–[8]: they define a set of modification rules to generate a space of patches and search in the space for patches that are *plausible* (i.e., the corresponding patched programs pass the test suite). A study by Long and Rinard [9] shows that (1) the search space, though huge, could be insufficient to contain a correct patch and (2) the search space often contains hundreds of plausible-but-incorrect patches which could simply block the finding of a correct one. Within a 12-hour time limit, the state-of-the-art repair techniques SPR [5] and Prophet [10] generated patches for less than 60% bugs in a dataset containing 69 bugs with more than 60% of the first found patches being incorrect. Early repair techniques are shown to have poor performance: as [7] shows, the majority of patches generated by GenProg [1], AE [2], and RSRepair [4] are incorrect.

To address the problem, the study [9] suggests leveraging repair information beyond the test suite to create a search space that is likely to contain a correct patch and is *targeted* so that the correct patch could be effectively identified. One idea is to leverage existing code fragments to produce effective patches. We call the code fragments that contain the correct forms of expressions, statements, etc. and can be used for generating a correct patch the *repair* code fragments. GenProg assumes the faulty program itself contains the repair code fragments at the statement level for patch generation. The study by Barr et al. [11] has demonstrated the feasibility of this assumption. If the repair code fragments may exist in the local program, they may also exist elsewhere in many non-local programs. The study by Sumi et al. [12] supports this assumption. They found up to 69% of the repair code fragments (in the form of code lines) can be obtained (possibly with identifier renaming) from both the local program and non-local programs. The study is based on the UCI dataset [13] containing 13,000 Java projects. We believe it is more likely to find the repair code fragments for bug[1] repair in smaller granularity (e.g., at the expression level) and from a larger code database (e.g., GitHub, which is huge and is still rapidly growing).

Repair code fragments could possibly exist in the faulty program itself and/or in non-local programs. Then the problem is how to find and leverage such code fragments to produce patches. One idea is to use semantic code search, i.e., finding code fragments that are likely to be semantically correct. However, semantic code search is often expensive and it may fail to find many repair code fragments that do not represent the correct implementation (they may contain more functional features than the correct implementation does, they may use different data types or side-effect processing mechanisms, etc.) but can be leveraged to produce a correct patch. CodePhage (CP) [14] and SearchRepair [15] are two repair techniques that use semantic code search. CP's code search relies on code execution, and it can only find code that can process the given inputs. SearchRepair uses symbolic execution to

[1]In this paper, we use "bug" and "fault" interchangeably.

encode program semantics as constraints. Symbolic execution, however, has limited expressive power for program semantics. SearchRepair's code search is based on constraint solving which is undecidable in general and is often expensive. Currently SearchRepair was only shown to work for small C programs.

If semantic code search is still limited, the natural question would be: does syntactic code search work? Our paper answered this question. We propose a novel repair technique ssFix which performs syntactic code search to find and leverage existing code fragments from a codebase (which consists of the local faulty program and an external code repository) to produce patches for bug repair. We assume a repair code fragment that can be effectively leveraged for bug repair to be syntax-related (i.e., structurally similar and conceptually related) to the fault-located part of the faulty program. Intuitively, such a repair code fragment is likely to implement a coding task similar to what is implemented in the faulty code fragment (e.g., both as iterating a list of data items to look for certain values having similar names) and implements it correctly. Compared to SearchRepair and CP, ssFix is not directly targeted at finding code fragments that are semantically correct. Instead, ssFix uses a lightweight syntactic code search (based on a Boolean model and a TF-IDF vector space model) to find syntax-related code fragments where a repair code fragment is likely to exist. Given such a fault-related code fragment (as a candidate code chunk), ssFix translates the code chunk by unifying the identifier names in it with those in the faulty code fragment (the target code chunk), matches the components between the two chunks, and produces patches for the target chunk based on the syntactic differences that exist between the matched components and in the unmatched components. For a candidate chunk that is syntax-related to the target chunk, the syntactic differences are small, and the search space is largely reduced. Through experiments, we demonstrated the feasibility of the assumption on which ssFix is built and the effectiveness of ssFix for bug repair.

In this paper, we make the following contributions:

- We developed a novel automated repair technique ssFix which performs syntactic code search to leverage existing code from a codebase to produce patches for bug repair. ssFix is currently available at *https://github.com/qixin5/ssFix*.
- We evaluated ssFix on all the 357 bugs in the Defects4J dataset. Our results show that ssFix successfully repaired 20 bugs with valid patches generated. The median time for producing a patch is about 11 minutes. Compared to five other repair techniques for Java, our results show that ssFix has a better performance.

II. Overview

In this section, we show an overview of ssFix and explain how it works with an example. ssFix accepts as input a faulty program, a fault-exposing test suite, and a codebase consisting of the faulty program and a code repository (we used the Merobase repository [16]). As output, ssFix either produces a patched program that passes the test suite or nothing if it cannot find one within a given time budget. ssFix goes through four stages to repair a bug: *fault localization*, *code search*, *patch generation*, and *patch validation*.

```
1  public static boolean isSameLocalTime(Calendar cal1,Calendar cal2){
2  if (cal1 == null || cal2 == null){
3  throw new IllegalArgumentException(``The date must not be null'');
4  }
5  return(cal1.get(Calendar.MILLISECOND)==cal2.get(Calendar.MILLISECOND)&&
6  cal1.get(Calendar.SECOND)==cal2.get(Calendar.SECOND) &&
7  cal1.get(Calendar.MINUTE)==cal2.get(Calendar.MINUTE) &&
8  cal1.get(Calendar.HOUR)==cal2.get(Calendar.HOUR) &&
9  cal1.get(Calendar.DAY_OF_YEAR)==cal2.get(Calendar.DAY_OF_YEAR) &&
10 cal1.get(Calendar.YEAR)==cal2.get(Calendar.YEAR) &&
11 cal1.get(Calendar.ERA)==cal2.get(Calendar.ERA) &&
12 cal1.getClass()==cal2.getClass());
13 }
```

Fig. 1. The faulty method of L21 (the fault is in red)

We use an example to go through the four stages. The faulty method as shown in Figure 1 is from a faulty program (bug id: *L21*) in the Defects4J bug dataset. It accepts as parameters two calendar objects `cal1` and `cal2` and checks whether they represent the same time. The fault is at Line 8 where the 12-hour calendar field `Calendar.HOUR` is used for comparing two local hours. Given two calendar objects whose hour fields are different (e.g., one is 4 and the other is 16) but all the other fields are identical, the faulty program may treat them as identical although they represent different times (one is early morning and one is late afternoon). For bug repair, the following modification should be made

```
- cal1.get(Calendar.HOUR) == cal2.get(Calendar.HOUR)
+ cal1.get(Calendar.HOUR_OF_DAY) == cal2.get(Calendar.HOUR_OF_DAY)
```

where the 24-hour calendar field `Calendar.HOUR_OF_DAY` is used. The modification is relatively simple, but none of the repair tools that ssFix is compared to succeeded for this bug. By leveraging existing code that is syntax-related to the bug context, ssFix successfully repaired the bug with the correct patch generated.

A. Fault Localization

In the first stage, ssFix employs the fault localization technique GZoltar (version 0.1.1) [17] to identify a list of suspicious statements in the program that are likely to be faulty. The statements in the list are ranked by their suspiciousness (measured as scores) from high to low. For bug repair, ssFix goes through the list: each time it looks at one statement (the *target* statement) and works on generating patches for a local code area (as a code chunk) including the statement (we will explain how to generate such a code chunk later). Currently, ssFix can only produce patches that make local changes (i.e., within the local code chunk) in the faulty program, though this may involve modifying more than one statement. (Note that if the faulty program crashed with a stack trace printed, ssFix will first follow the stack trace to look at each statement in the stack trace first if the statement is from the faulty program. GZoltar does not use the stack trace information to compute a statement's suspiciousness. To repair a failure which causes the program to crash, we assume the statements from the stack trace are more suspicious than the other statements in the faulty program.) The faulty return statement starting at Line 5

```
GregorianCalendar calEnd = new GregorianCalendar();
calEnd.setTimeInMillis(end.getTime());
if (calStart.get(Calendar.HOUR_OF_DAY)==calEnd.get(Calendar.
      ↪ HOUR_OF_DAY)
&& calStart.get(Calendar.MINUTE)==calEnd.get(Calendar.MINUTE)
&& calStart.get(Calendar.SECOND)==calEnd.get(Calendar.SECOND)
&& calStart.get(Calendar.MILLISECOND)==calEnd.get(Calendar.
      ↪ MILLISECOND)
&& calStart.get(Calendar.HOUR_OF_DAY)==0
&& calStart.get(Calendar.MINUTE)==0
&& calStart.get(Calendar.SECOND)==0
&& calStart.get(Calendar.MILLISECOND)==0
&& start.before(end))
return true;
```

Fig. 2. A candidate code chunk retrieved from the Merobase repository (the fix expression is in purple). The chunk's enclosing method *isAllDay* checks whether the two time values obtained by *start.getTime()* (not shown) and *end.getTime()* both as millisecond values) represent the starting time of two days (from 00:00 of one day to 00:00 of the next day). The full class name of the chunk is *org.compiere.util.TimeUtil*.

in Figure 1 is the second statement ssFix looks at among all the suspicious ones.

B. Code Search

Given a target statement identified as suspicious, ssFix goes through three steps to find syntax-related code fragments from the codebase: *target chunk identification*, *token extraction*, and *candidate retrieval*. As the first step, ssFix generates a code chunk $tchunk$ including the statement itself and possibly its context. ssFix then searches for code fragments in the codebase as $cchunk$s that are syntax-related, i.e., structurally similar and conceptually related, to $tchunk$. A $tchunk$ to be used as the query for code search should not be too small (e.g., including only a simple statement as *return x*) because it does not include enough context. On the other hand, it should also not be large. The study by Gabel and Su [18] shows that a code fragment with more than 40 tokens can be too unique in general to have similar code fragments retrieved for code search at the repository level. Based on this result, we develop a simple chunk generation algorithm (Algorithm 1) to generate a $tchunk$ including the target statement and its local context if the statement is not too large (to determine its size, we use a threshold based on the LOC of the statement). For our example, ssFix uses this algorithm to produce a $tchunk$ with only the return statement included.

As the second and third steps, ssFix extracts the structural k-gram tokens and the conceptual tokens from $tchunk$ and invokes the Apache Lucene search engine [19] to do a document search to obtain a list of indexed code fragments (treated as documents) from the codebase. The retrieved list of code fragments (which we call the *candidate* code chunks, or $cchunk$s) are ranked from the ones that are the most syntax-related to $tchunk$ to the least (measured by the scores computed by Lucene's default TF-IDF model from high to low). Later, ssFix goes through the list and leverages each $cchunk$ to produce independent patches for $tchunk$. More details can be found in Section III-A. The retrieved $cchunk$ shown in Figure 2 is what ssFix later uses to produce a correct patch for $tchunk$. This $cchunk$ is ranked No. 6 among all the retrieved chunks.

C. Patch Generation

ssFix leverages a candidate chunk $cchunk$ to produce patches for $tchunk$ in three steps: *candidate translation*, *component matching*, and *modification*. $tchunk$ and $cchunk$ may use different identifier names for variables, fields, types, and methods that are syntactically (and semantically) related. For example, the two chunks in Figure 1 and in Figure 2 use different names: `cal2` and `calEnd` for a related variable. As the first step, ssFix translates $cchunk$ (if retrieved from a non-local program) by unifying the identifier names in $cchunk$ with those that are syntactically related in $tchunk$. Without such a translation, ssFix would often fail to directly use statements and expressions from $cchunk$ to produce patches for $tchunk$: the patched program could simply fail to compile for using unrecognized names. We developed an heuristic algorithm (Algorithm 2) which ssFix uses to match variables, fields, types, and methods between $tchunk$ and $cchunk$ based on how they are used in the two chunks. ssFix then renames the variables, fields, types, and methods in $cchunk$ to their matched counterparts in $tchunk$ to achieve the translation. For our example, ssFix determines `calStart` to match `cal1` and `calEnd` to match `cal2` based on pattern-matched expressions like the following three pairs (see Section III-B1 for more details).

```
cal1.get(Calendar.MILLISECOND) == cal2.get(Calendar.MILLISECOND)
calStart.get(Calendar.MILLISECOND) == calEnd.get(Calendar.MILLISECOND)

cal1.get(Calendar.SECOND) == cal2.get(Calendar.SECOND)
calStart.get(Calendar.SECOND) == calEnd.get(Calendar.SECOND)

cal1.get(Calendar.MINUTE) == cal2.get(Calendar.MINUTE)
calStart.get(Calendar.MINUTE) == calEnd.get(Calendar.MINUTE)
```

ssFix creates a translated version of $cchunk$ as $rcchunk$ by renaming the two variables `calStart` and `calEnd` to their respective matched ones `cal1` and `cal2` in $tchunk$.

The translated chunk $rcchunk$ may not represent the correct patch but may contain the correct forms of components (expressions and statements) to be used in $tchunk$ or indirectly suggest a faulty statement in $tchunk$ to be deleted for producing a correct patch. Instead of replacing $tchunk$ with $rcchunk$ at the chunk level for patch generation, ssFix matches components that are syntactically related between the two chunks and produces patches based on the syntactic differences that exist between the matched components and in the unmatched components. Specifically, ssFix uses a modified version of the tree matching algorithm used by ChangeDistiller [20] to do component matching, and it modifies $tchunk$ to produce patches using three types of operations: *replacement*, *insertion*, and *deletion* (see Section III-B2 and Section III-B3). For our example, ssFix found the following pair of components (and 26 others) from $tchunk$ and $rcchunk$ to match.

```
cal1.get(Calendar.HOUR) == cal2.get(Calendar.HOUR)
cal1.get(Calendar.HOUR_OF_DAY) == cal2.get(Calendar.HOUR_OF_DAY)
```

In $tchunk$, it then replaces the first component with the second (from $rcchunk$) to produce the correct patch.

D. Patch Validation

For each $cchunk$, ssFix produces a set of patches. It filters away patches that are syntactically redundant and patches that have been tested earlier (generated by other *cchunk*s). ssFix next sorts the filtered patches based on the modification types and the modification sizes to make a correct patch likely to be found before an overfitting patch (such a patched program can pass the test suite but does not actually or fully repair the bug). More details can be found in Section III-C. ssFix reports the first patched program that passes the test suite. If no such program can be found, ssFix looks at the next *cchunk* from the retrieved list and repeats the patch generation and patch validation processes. For our example, ssFix successfully found the correct patch after validating 202 individual patched programs that failed the testing (the majority of those simply failed the fault-exposing test case). It took ssFix less than seven minutes to find this patch.

III. Methodology

In this section, we elaborate on the last three stages that ssFix takes for bug repair. Fault localization itself is a research field and is not the focus of this paper. ssFix simply uses the approach described in Section II-A to do fault localization.

A. Code Search

The code search stage of ssFix starts with a target statement s identified as suspicious in the first stage. ssFix generates a local code chunk $tchunk$ including s itself and possibly the local context of s. ssFix then extracts the structural and conceptual tokens from the text of $tchunk$. ssFix treats the extracted tokens as a vector of terms and uses Lucene's Boolean Model and its TF-IDF vector space model to find candidate code chunks $cchunk$s that are syntax-related to $tchunk$ from the codebase.

Algorithm 1 Generating a Local Target Code Chunk

```
Input: $s$, $th$                          ▷ $s$: target statement, $th$: LOC (we use 6)
Output: $tchunk$                          ▷ A target code chunk
 1: function ChunkGen($s$,$th$)
 2:   $tchunk \leftarrow \{s\}$
 3:   if $getSize(tchunk) \geq th$ then return $tchunk$
                                          ▷ $getSize$ returns the LOC of a code chunk
 4:   $s_0 \leftarrow$ get the parent statement of $s$
 5:   if $s_0$ exists then
 6:     $tchunk_0 \leftarrow \{s_0\}$
 7:     if $getSize(tchunk_0) \leq th$ then return $tchunk_0$
 8:   $s_1 \leftarrow$ get the statement before $s$ in its block
 9:   $s_2 \leftarrow$ get the statement after $s$ in its block
10:   if both $s_1$ and $s_2$ exist then
11:     $tchunk_1 \leftarrow \{s_1, s, s_2\}$
12:     if $getSize(tchunk_1) \leq th$ then return $tchunk_1$
13:   else if $s_1$ exists but $s_2$ does not exist then
14:     $tchunk_2 \leftarrow \{s_1, s\}$
15:     if $getSize(tchunk_2) \leq th$ then return $tchunk_2$
16:   else if $s_1$ does not exist but $s_2$ exists then
17:     $tchunk_3 \leftarrow \{s, s_2\}$
18:     if $getSize(tchunk_3) \leq th$ then return $tchunk_3$
19:   else
20:     return $tchunk$
```

1) Chunk Generation: A $tchunk$ with some context of s included could provide information about what s intends to do with the semantics potentially common to a large amount of existing code fragments in the codebase. Although it is often necessary to include some context of s (especially when s is too simple in its form as *return x* for example, and does not contain enough semantics), it can be a bad idea to include a large context (e.g., a method that implements multiple tasks). As the study [18] shows, for repository code search, significant syntactic redundancies were observed for code containing only up to 40 tokens (or 5-7 lines approximately). A larger code fragment is likely to be too unique. Based on this observation, we developed a simple algorithm *chunkgen* which generates a $tchunk$ including only the local context of s with a chunk-size threshold th (6 LOC) specified. As shown in Algorithm 1, if the size of s is equal to or larger than th, ssFix simply produces a $tchunk$ including s itself (Lines 2-3). Otherwise, ssFix produces a $tchunk$ including either the enclosing parent statement of s up to the declared method (not inclusive), if any exists, (Lines 5-7) or a maximum of s and its two neighboring statements (Lines 8-18) as long as the size of $tchunk$ is no larger than th.

The way ssFix generates $cchunk$s is similar: For each Java source file in the codebase, ssFix looks at every method defined in every class defined in the file. It extracts the following code fragments within the method as $cchunk$s: (1) every compound statement which contains children statements and (2) every sequential three statements within each code block (e.g., a body block of a for-statement). (Note that for any compound statement which has a non-block single statement as its body, ssFix will create a new block as the body containing the statement. Also note that if a code block contains no more than three statements, all the statements are then included in the chunk). ssFix produces a $cchunk$ using (1) and (2) to cover the two cases it produces a $tchunk$ using the target statement's parent statement and the target statement itself plus its neighboring statements. Note that ssFix does not use any chunk-size threshold to produce a $cchunk$. This makes ssFix be able to find a $cchunk$ that is smaller or larger than $tchunk$ (for statement deletion and insertion).

Currently, ssFix produces a relatively small $tchunk$ (with only the suspicious statement s and possibly its local context included) used for both code search and patch generation. Our experiments show that this works reasonably well. But it is still possible to use a larger $tchunk$ including more than the local context of s to do code search (possibly with different query weights put on different context levels) to have a $cchunk$ in a comparable size retrieved, and later to find smaller chunks (e.g., using clone detection techniques like [21], [22]) within the original two chunks for patch generation. We consider exploring this as our future work.

2) Token Extraction: Given either $tchunk$ or $cchunk$, ssFix extracts the structural k-gram tokens and the conceptual tokens from the text of the chunk. For every generated $cchunk$ in the codebase, ssFix employs Lucene [19] to create an index for the extracted tokens to facilitate code search. Given $tchunk$, ssFix searches in the codebase for $cchunk$s that have "similar" tokens using Lucene's Boolean model and its TF-IDF vector space model.

Extracting the structural k-gram tokens: ssFix first tokenizes the text of a chunk and gets a list of tokens. To mask names, number constants, and literals that are program specific, ssFix symbolizes different types of tokens: ssFix uses the symbol \$v\$ for non-JDK variables and fields, \$t\$ for non-JDK & non-primitive types, \$m\$ for non-JDK methods, \$lb\$ for boolean literals (*true* or *false*), \$ln\$ for number constants, and \$ls\$ for string literals that contain whitespace characters (e.g., as an exceptional message). ssFix does not symbolize JDK tokens, primitive types, character literals, or string literals that do not contain whitespace characters since they are often semantics-indicative. We call the symbolized tokens the *code pattern tokens* and we call the string of these tokens concatenated by single spaces the *code pattern*. ssFix next splits the list of code pattern tokens into sub-lists by curly brackets and semicolons to avoid generating k-grams that are not very interesting (e.g., a k-gram that starts at the end of one statement but ends at the start of another). Finally, ssFix concatenates (with no space in between) every sequential k (we set k=5) tokens within every sub-list of tokens to get the structural k-gram tokens. (Note that if a sub-list contains less than k tokens, ssFix would produce a less-than-k-gram token.) Given a statement as `str.charAt(1)=='e';` where `charAt` is a JDK method, ssFix splits the statement into a list of tokens, symbolizes the tokens (changing `str` to `$v$` and `1` to `$ln$`), splits the symbolized list into a sub-list of tokens by semicolon, and finally gets a list of four 5-gram tokens: {`$v$.charAt($ln$`, `.charAt($ln$)`, `charAt($ln$)==`, `($ln$)=='e'`}.

Extracting the conceptual tokens: Two chunks that are conceptually related often use common tokens such as "time", "iterator", or "buffer". ssFix extracts such conceptual tokens as follows: ssFix first tokenizes the text of a chunk and gets a list of tokens containing *Java identifiers* only. For any token that is camel-case or contains underscores or numbers, 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 and the Java keywords. For example, the list of conceptual tokens for `str.getChars(0,strLen,buffer,size)` is {"str", "getchars", "chars", "strlen", "str", "len", "buffer", "size"} (Note that "get" is a stop word that is eliminated).

3) Candidate Retrieval: For candidate retrieval, ssFix invokes Lucene's query search with the query tokens being the extracted tokens from $tchunk$[2]. It uses Lucene's default TF-IDF vector space model (which uses *Lucene's Practical Scoring Function* defined in [23]) to retrieve $cchunk$s. The retrieval process ignores any $cchunk$ whose the number of matched tokens (the tokens that are matched with those in $tchunk$) is less than $n/8$ where n is the total number of tokens in $tchunk$. To do so, ssFix uses Lucene's Boolean model.

For each $tchunk$, ssFix obtains a list of $cchunk$s that have the highest relatedness scores ranked from high to low. Currently, it only looks at the top 100 (at most) $cchunk$s that are not syntactically redundant for bug repair.

[2]It is also possible to invoke Lucene's query search twice using the structural tokens and the conceptual tokens independently and then merge the results, but we did not experiment this.

B. Patch Generation

In this stage, ssFix leverages a candidate chunk $cchunk$ to produce patches for $tchunk$ in three steps: *candidate translation*, *component matching*, and *modification*.

Algorithm 2 Creating an Identifier Mapping

```
Input: tchunk, cchunk
Output: imap[idBind → idBind]                ▷ idBind is an identifier binding
 1: imap[idBind → idBind] ← empty
 2: cmap[(idBind, idBind) → int] ← empty
 3: tcompts, ccompts ← get the component lists of tchunk, cchunk (components
    visited in pre-order)
 4: matched_compts ← match components between tcompts and ccompts by
    code pattern equality
 5: for all (tcompt, ccompt) ∈ matched_compts do
 6:   tptokens, cptokens ← get the code pattern tokens of tcompt, ccompt
 7:            ▷ tptokens & cptokens are two lists having identical elements
 8:   for all (tptoken, cptoken) ∈ (tptokens, cptokens) at every list index do
 9:     if tptoken and cptoken are both identifier symbols then
10:       tidbind ← get the identifier binding of tptoken
11:       cidbind ← get the identifier binding of cptoken
12:       if (cidbind, tidbind) is an entry in cmap then
13:         c ← cmap.get(cidbind, tidbind)
14:         cmap.add((cidbind, tidbind), c + 1)
15:       else
16:         cmap.add((cidbind, tidbind), 1)
17: for all cidbind from cmap do
18:   tidbind ← get the mapped identifier with the max value of c (tie breaking by
      the Levenshtein Similarity between identifier strings)
19:   imap.add(cidbind, tidbind)
20: return imap
```

1) Candidate Translation: A candidate chunk $cchunk$ and the target chunk $tchunk$ may use different identifier names for variables, fields, types, and methods that are syntactically and semantically related, especially when they are not from the same program. We developed an heuristic algorithm shown in Algorithm 2 to map variable, field, type, and method identifiers appeared in $cchunk$ to those in $tchunk$ that are syntactically related (and may thus be semantically related) based on matching the code patterns of their contexts. (The code pattern used here is identical to what we defined in Section III-A2 but with all non-JDK identifiers, number constants, and literals symbolized to increase matching flexibility). Given a $cchunk$ that is not from the local, faulty program (where $tchunk$ is from), ssFix uses the algorithm to match their identifiers and renames every identifier in $cchunk$ (which has a match) as its matched identifier in $tchunk$ to get a translated version of $cchunk$ as $rcchunk$. Since a $cchunk$ and a $tchunk$ from the same faulty program use identifier names consistently, ssFix does not create a translated version for such a $cchunk$.

Algorithm 2 accepts as input $tchunk$ and $cchunk$. It outputs an identifier mapping $imap$ that maps an identifier that appears in $cchunk$, as a reference binding (or a binding), to an identifier that appears in $tchunk$, also as a binding. (ssFix matches and renames all identifiers that have the same binding consistently.) The algorithm starts by collecting in pre-order a list of components (statements and expressions) in the tree structure of the chunk (for either $tchunk$ or $cchunk$) that are not number constants, literals (*boolean*, *null*, *character*, and *string* literals), or identifiers (Line 3). These components represent all the contexts of all the identifiers in the chunk. The algorithm then matches the components (Line 4) by comparing their code patterns. Two components can match iff their code

TABLE I
SUB-COMPONENT REPLACEMENT RULES FOR CERTAIN TYPES OF MATCHED COMPONENTS

Component	**Rule**
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 initializers, condition, & updaters 5. Replace body
Loop Statements (not both as for-statements)	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
Synchronized Statements	1. Replace synchronized expression 2. Replace body
Return Statements (with *boolean* returned expressions)	1. Combine the expressions with $\&\&$ 2. Combine the expressions with $\|\|$
Catch Clauses	1. Replace caught exception 2. Replace body
Assignments/Infix Expressions	1. Replace left-hand side 2. Replace operator 3. Replace right-hand side
Method Calls/Super Method Calls	1. Replace caller expression 2. Replace method name 3. Replace arguments$*$
Constructor Calls (i.e., this(...))	1. Replace arguments$*$
Super Constructor Calls	1. Replace caller expression 2. Replace arguments$*$
Prefix/Postfix Expressions	1. Replace operator 2. Replace operand

$*$ ssFix may produce multiple patches by replacing each individual argument of $tcpt$ with the corresponding argument of $ccpt$ in the same argument index. This only happens when the two components have the same number of arguments.

patterns (as two strings) are identical. For every matched components whose code patterns are identical (and thus share an identical list of code pattern tokens), the algorithm obtains the two lists of code pattern tokens (Line 6). At every index where the two code pattern tokens are both identifiers, the algorithm gets the identifier bindings, matches them, and saves this match with a count in a map $cmap$ (Lines 8-16). Finally, the algorithm iterates $cmap$, for each identifier binding in $cchunk$ ($cidbind$), it finds its matched identifier binding in $tchunk$ ($tidbind$) with the maximum matching count. If there are more than one such matched $tidbind$s, the algorithm breaks the ties by comparing the string similarity of the identifier bindings (Lines 17-19).

2) Component Matching: ssFix matches components between $tchunk$ and $rcchunk$ to identify their syntactic differences at the component level. Later it leverages the syntactic differences that exist between the matched components and in the unmatched components to produce patches for $tchunk$. ssFix extends the tree matching algorithm of ChangeDistiller (Fig. 9 in [20]) to do component matching based on the component types, structures, and contents. The original algorithm performs tree matching at the statement level and is used for code evolutionary analysis. The algorithm used by ssFix follows its basic idea to match leaf nodes first (using the $match_1$ function) and then inner nodes (using the $match_2$ function) in a bottom-up way. We make changes to the original algorithm on the definitions of leaf and inner nodes, node compatibility, and node similarity.

Specifically, we define a *leaf* node to be either a simple statement which has no children statements or an expression that is not a number constant, a literal, or an identifier. We define an *inner* node to be a compound statement that has children statements. We give a new definition for the node compatibility (the l function in [20]) as follows: (1) a leaf node is not compatible with an inner node, (2) two leaf nodes are compatible if (a) their node types are equal (e.g., both as return statements) and (b) they follow the node-type-specific rules: for *ArrayAccess* or *ArrayCreation*, the array types should be compatible (i.e., the array dimensions are equal and the element types are equal); for *ClassInstanceCreation*, the class types should be identical; for *InfixExpression*, *PostfixExpression*, or *PrefixExpression*, the expression operators should be identical; for *Assignment*, the assignment operators should be identical; and for *MethodInvocation*, the method names should be identical, and (3) two inner nodes are compatible if their node types are equal or they are both loop statements (for, while, or do statements). As for the node similarity metrics used in the $match_1$ and $match_2$ functions in [20], we make two changes: (1) we decrease the values of the thresholds f and t and (2) we ignore the bigram string similarity part for the similarity metric in $match_2$. ChangeDistiller was designed to match nodes that are highly similar for evolutionary analysis. In our context, we decrease the thresholds f and t to allow components that are syntactically related but are not highly similar to match. Currently ssFix uses 0.2 for f and 0.4 for t and it works reasonably well with these thresholds for our experiments. We do not consider the similarity of two conditions (as if-conditions or loop-conditions) as a factor to match two compound statements (as two inner nodes) because a bug could make one condition dissimilar to the other. In such case, we still allow the two statements to match as long as they have similar children according to the Dice Coefficient similarity used in the $match_2$ function in [20] so that the faulty condition has a chance of being repaired.

3) Modification: In the final step of patch generation, ssFix modifies $tchunk$ based on the matched and unmatched components between $tchunk$ and $rcchunk$ to yield an initial set of patches using three types of modifications: *replacement*, *insertion*, and *deletion*. We next discuss each in turn.

Replacement: For every matched components ($tcpt$, $ccpt$) where $tcpt$ is a component from $tchunk$ and $ccpt$ is a component from $cchunk$, ssFix replaces $tcpt$ with $ccpt$ and the sub-components of $tcpt$ with the sub-components of $ccpt$ to produce patches. Specifically, ssFix first replaces $tcpt$ with $ccpt$ to produce a patch if $tcpt$ is not syntactically identical to $ccpt$. ssFix may do more replacements on the sub-components of $tcpt$ and $ccpt$ based on their types following the rules we created in Table I. (Recall that if $tcpt$ matches $ccpt$, either they have the same component type or they are both loop statements.) For each row in Table I, there is more than one rule. ssFix follows the rules to produce patches independently: each time, it follows one rule to produce one patch (it would not produce a patch if the replacement makes no actual syntactic changes.) Note that ssFix may make multiple changes using one replacement. For example, it may follow Rule 2 for loop statements to replace a loop body with another which may make changes to several statements within the body.

Insertion & Deletion: ssFix inserts an unmatched statement component ($cstmt$) from $rcchunk$ in $tchunk$ to produce a patch. For any component ($cstmt$) in $rcchunk$, ssFix first checks whether it is qualified for insertion, i.e., (1) whether it is a statement that has no match in $tchunk$ and (2) whether it has no matched children statements. If not qualified, ssFix ignores the insertion of $cstmt$ because the potential occurrence of $cstmt$ in $tchunk$ could lead to statement redundancy caused by itself or by its children statements. If $cstmt$ is qualified, ssFix computes estimated positions where $cstmt$ is likely to fit in $tchunk$ and later inserts $cstmt$ at every estimated position to yield patches. Specifically, ssFix first finds the two sibling statements of $cstmt$ in its parent block (as csl and csh) that are closest to $cstmt$ and have matches. ssFix gets their matched statements tsl and tsh in $tchunk$ and inserts $cstmt$ at every position in between (if they both exist and are from the same block). Otherwise, if at least one of tsl and tsh exists, ssFix inserts $cstmt$ at every position after tsl in its located block and/or at every position before tsh in its located block to yield patches. If neither tsl nor tsh exists, ssFix ignores the insertion for $cstmt$ since there is no matching evidence that $cstmt$ is needed.

For deletion, ssFix deletes any statement component in $tchunk$ that has no matched statement in $rcchunk$. Similar to insertion, if the unmatched statement has matched children statements, ssFix ignores its deletion.

C. Patch Validation

ssFix leverages a $cchunk$ retrieved from the codebase to produce a set of patches for $tchunk$. In this stage, ssFix first filters aways patches that are syntactically redundant and patches that have been tested earlier (generated by other $cchunk$s), next sorts the patches by the modification types and sizes, then validates each patched program against the test suite, and finally reports the first one (if any) that passes the test suite. Like every repair technique that uses a test suite as the correctness criterion for patch evaluation, it is possible that ssFix produces an overfitting, patched program that passes the test suite but does not actually or fully repair the bug. Studies [7], [24] have shown that (1) a repair technique is more likely to produce an overfitting patch using deletion than using other types of modifications and (2) a simple patch is less likely to be overfitting than a complex patch. Based on these results, we created the following rules to rank two patches: (1) a patch generated by replacement or insertion *always* has a higher rank than a patch generated by deletion, (2) if there is a tie, a patch with a smaller modification tree height has a higher rank, and (3) if there is still a tie, a patch with a smaller edit distance has a higher rank. We define a modification tree height of a patch to be the maximum heights of the tree structures of the modification-related components cpt and cpt' (for insertion or deletion, if a component is *null*, the height is 0). We define an edit distance of a patch to be the edit distance between the content strings of cpt and cpt' (if a component is *null*, the content string is empty). ssFix follows the rules to rank patches and does patch sorting.

Note that ssFix may produce hundreds of patches given a $cchunk$ that is dissimilar to $tchunk$. For efficiency, ssFix only selects a maximum of the top-sorted k (we set $k = 50$) patches that it produced for $tchunk$ using a $cchunk$ for validation. If a patched program compiles, ssFix first tests it against the test case(s) that the original, faulty program failed. If the patched program succeeds, ssFix then tests it against the test suite.

IV. Empirical Evaluation

To evaluate the performance of ssFix, we used the Defects4J bug dataset (version 0.1.0) [25] which contains a set of 357 real bugs. We ask two research questions.

- **RQ1**: How many bugs can ssFix repair? What is the performance of ssFix on repairing these bugs?
- **RQ2**: Compared to other techniques, how effective is ssFix?

We conducted two experiments to answer them. We next show each experiment in turn.

A. RQ1

We implemented ssFix and evaluated its performance on all 357 real bugs in the Defects4J bug dataset. Our results show that ssFix repaired 20 bugs with valid patches generated. The median time for generating a plausible patch is about 11 minutes.

1) Experimental Setup:

a) Bug Dataset: The Defects4J dataset [25] consists of 357 real bugs from five Java projects: *JFreeChart* (C), *Closure Compiler* (Cl), *Commons Math* (M), *Joda-Time* (T), and *Commons Lang* (L). Each bug in the dataset is associated with a developer patch showing how the bug can be correctly repaired. The dataset has been commonly used for evaluating an automated repair technique for Java [26], [27], [8], [28].

b) ssFix's Running Setup: Our implementation of ssFix used the Merobase repository [16] (which contains about 2.5 million Java source files, or about 180 million LOC) as the external code repository and five versions of the projects (C8, Cl14, L6, M33, and T4) as the local faulty programs[3]. To avoid using a fixed version of a bug to produce patches, in the code search stage, ssFix ignores any candidate chunk $cchunk$ retrieved from the codebase if (1) the full-class name of $cchunk$'s located class is the same as that of the target chunk's (or $tchunk$'s) located class[4] and (2) the signature of $cchunk$'s enclosing method is the same as that of $tchunk$'s method.[5] We ran ssFix to repair each bug within a time budget of 120 minutes on machines with eight AMD Phenom(tm) II processors and 8G memory.

[3]For each of the three bugs: M53, M59 and M70, the fault's located class contains a repair statement. The repair statement, however, is not contained in the class of M33 whose class name is identical to that of the fault's located class. So we additionally indexed the fault's located class for each bug (three classes in total).

[4]The *Commons Lang* & the *Commons Math* projects may use either *lang3* & *math3* or *lang* & *math* as parts of their package names respectively. We unified these name differences for comparing two class names.

[5]Even doing so, we still manually found, in our initial experiments, the two *cchunk*s (for *L43* and *L33*) that ssFix used to yield patches are suspicious to be from the fixed versions of the two faulty programs. We created a black-list for the enclosing methods of those *cchunk*s.

TABLE II
ALL PLAUSIBLE PATCHES GENERATED BY SSFIX

Project (#Bugs)	Time (in minutes)			#Plausible	#Valid*	#Correct Sem(Syn)*	CChunk Rank			CChunk Locality		#Tested Patch		
	min	max	med				min	max	med	#Local	#Non-Local	min	max	med
JFreeChart (26)	3.4	77.9	8.8	7	3	2(2)	1	65	34	2	5	2	4337	132
Closure Compiler (133)	7.6	34.8	12.8	11	2	1(1)	1	51	1	9	2	2	489	84
Commons Math (106)	2.2	100.5	14.8	26	10	7(6)	1	91	7	12	14	1	5609	171.5
Joda-Time (27)	1.8	8.1	4.0	4	0	0(0)	1	24	5	1	3	3	426	61.5
Commons Lang (65)	3.4	56.4	6.1	12	5	5(5)	1	60	8	1	11	3	2454	34.5
Total (357)	1.8	100.5	10.7	60	20	15(14)	1	91	6.5	25	35	1	5609	99

* We manually compared a generated patch to the developer patch to determine its validity and correctness.

2) Results: Table II is a summary[6] of the repair performance of ssFix. From left to right, the table shows the project name and the number of bugs in the project, the repairing time (min, max, and median), the number of bugs for which plausible patches were generated (a patch is *plausible* if the patched program passes the test suite), the number of valid patches generated (we consider a patch to be *valid* if the patched program passes the test suite and does not introduce regressions in general), the number of correct patches generated (we consider a patch to be *correct* if it is semantically equivalent to the developer patch associated with the bug, in a stricter case, such a patch can be syntactically equivalent to the developer patch), the ranks (in min, max, and median from 1 to 100) of the candidate chunks used for generating the patches, the numbers of chunks retrieved from the local project and from the external code repository, and the number of failed patches ssFix created and tested (against at least one test case) before finding a plausible patch.

As shown, ssFix produced plausible patches for 60 bugs in total (a patch is plausible if the patched program passes the test suite). The running time (in minutes) for repairing these 60 bugs ranges from 1.8 to 100.5 with the median being 10.7. A plausible patch is produced and identified by ssFix automatically. To determine whether a generated, plausible patch does not introduce regressions and whether it is semantically correct in general, we manually compared the patch with the corresponding developer patch contained in the Defects4J dataset. Among the 60 plausible patches generated (for the 60 bugs), we determined 20 patches to be valid. Among the 20 valid patches, we determined 15 patches to be semantically equivalent to the developer patches associated with their repaired bugs, and 14 of the 15 patches to be not only semantically but also syntactically equivalent to the corresponding developer patches. In terms of passing the test suite without introducing regressions in general, we determined 5 patches to be valid though they are not semantically equivalent to the developer patches. Below is one such patch generated for the bug *M57*:

```
+ double sum=0; (by developer)
+ float sum=0; (by ssFix)
- int sum=0;
```

ssFix patched the program by changing the declared type of *sum* from *int* to *float* to avoid precision loss. The patched program now passes the test suite. Although the patch is not semantically equivalent to the developer patch, we consider it as valid. We manually determined 7 of the 60 plausible patches to be *defective* (and thus overfitting): they introduce regressions to their original programs and are thus invalid and incorrect. For four of them, ssFix deleted the expected program semantics. For the remaining 33 (60-20-7) patches, it is not easy for us to manually determine their validity since the patches are not syntactically equivalent or similar to the developer patches, so we released them for other reviews. All the 60 plausible patches and the corresponding candidate chunks can be found under *https://github.com/qixin5/ssFix/tree/master/expt0*. For each of the 20 valid patches, we provided an explanation as to why we believe it is valid/correct.

ssFix failed 297 (357-60) bugs with no patches generated. To understand the failures, we manually examined the developer patches for all the 357 bugs and found that there are 263 complex bugs for which the correct patches are not within the search space of ssFix (recall that ssFix can currently only repair relatively simple bugs by making modifications within a relatively small code chunk). Among the 297 failed bugs, there are 221 such complex bugs for which ssFix cannot produce *correct* patches. (But note that ssFix did produce *valid* patches for 2 of the 263 complex bugs: Cl115 & M30. Each such patch makes and only makes some but not all of the changes made by the developer patch, and the corresponding patched program passes the test suite.)

Among the other 94 (357-263) simple bugs, ssFix produced plausible patches for 33 bugs, and it failed 61 bugs with no patches generated. One challenge lies in the accuracy of fault localization. We found GZoltar simply failed to identify the target faulty statements for 15 bugs (among the 61 failed ones). We also found there are 19 bugs for which the suspicious ranks of the target statements are greater than 50 (with the median rank being 159), and ssFix did not actually looked at any of these target statements under the current running setup.

Another challenge lies in ssFix's code search ability in finding effective candidates. The current way ssFix does code search is not effective for all cases. The bug Cl10 is one example. ssFix produces a target chunk as shown below.

```
if (recurse) {
  - return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
  + return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
} else { return mayBeStringHelper(n); }
```

Since all the identifier names are locally defined by the faulty program, ssFix creates a code pattern with all the names symbolized, and extracts a list of structural tokens that are a little too general (which roughly say that the code chunk contains an if-statement and two method calls to be returned). The extracted conceptual tokens together are a little too unique

[6]The complete result can be found at https://github.com/qixin5/ssFix/blob/master/expt0/rslt.

to be used for finding related candidate chunks in the codebase. As a result, ssFix failed to find candidate chunks that are truly syntax-related from Merobase. The candidate chunks found from the local program however do not contain the correct expression to be used for bug repair. So ssFix failed to repair the bug. In the last part of Section III-A1, we propose a way for improvement and consider to explore it as our future work.

In principle, the ways ssFix uses to do candidate translation, component matching, and modification can also limit ssFix from producing a valid/correct patch. But we found these are not actual problems when a target statement is accurately located and an effective candidate chunk is found.

Since ssFix uses a test suite (as opposed to a formal specification) as the correctness criterion for patch evaluation, it can generate a defective patch which introduces regressions. An inaccurate fault localization technique and an ineffective candidate could both lead to a defective patch being generated. We actually found that it can be problematic to produce patches by deletion using a candidate chunk that is not very related to the target chunk. ssFix produced four defective patches by deleting the non-buggy statements.

B. RQ2

We compared ssFix to five other repair techniques for Java: jGenProg [26], jKali [26], Nopol (version 2015) [29], HDRepair [30], and ACS [31] on the same dataset. Compared to these techniques, our results show that ssFix has a better performance: it produced larger numbers of patches that are valid and correct with the efficiency of producing a plausible patch being either comparable or better. Note that we did not compare ssFix to other repair techniques that are written for C (e.g., SearchRepair [15], CodePhage [14], SPR [5], Prophet [10], and Angelix [32]) or are not publicly available as of August, 2017 (e.g., PAR [3]).

1) Experimental Setup: We ran jGenProg, jKali, Nopol, HDRepair, and ACS each to repair all the 357 bugs in the Defects4J dataset on machines that have the same configurations with the ones on which we ran ssFix. The time budget for repairing a bug is two hours (the same for ssFix). Since jGenProg and HDRepair use randomness for patch generation, we ran the tool (either jGenProg or HDRepair) in three trials[7] to repair a bug, and we considered the tool to have a valid/correct patch generated if it did so in at least one trial. For the other three techniques, we ran them each only in one trial to repair each bug.

2) Results: Table III shows the repairing time (min, max, and median) and the numbers of plausible, valid, and correct patches generated by all the six techniques. Figure 3 shows the ids of the bugs for which the techniques produced valid patches. Our results show that ssFix significantly outperforms jGenProg, jKali, and Nopol: ssFix produced many more valid

[7]Note that our experiment was very expensive and we only ran jGenProg/HDRepair in three trials. We believe our current running setup is sufficient to show that ssFix outperforms the two tools: the number of valid and correct patches generated by ssFix in one trial is about four times larger than the number of those patches generated by jGenProg or HDRepair in three trials.

TABLE III
ALL PLAUSIBLE PATCHES GENERATED BY SSFIX AND FIVE OTHER TECHNIQUES (SEE FIGURE 3 FOR THE SPECIFIC BUGS FOR WHICH VALID PATCHES WERE GENERATED BY THE SIX TECHNIQUES)

Tool	Time (in minutes)			#Plausible	#Valid	#Correct	
	min	max	med			sem	syn
ssFix	1.8	100.5	10.7	60	20	15	14
jGenProg	10.8	78.5	30.5	19(27)	3	3(5)	2
jKali	4.4	81.6	8.5	18(22)	1	1(1)	1
Nopol	1.6	101.3	12.6	33(35)	0	0(5)	0
HDRepair	8.2	87.7	52.3	16(23†)	5	4(10†)	3
ACS	88.8	113.1	93.9	7(23‡)	3	3(18‡)	2

The numbers in parentheses (in the 5th and 7th columns) are copied from the results reported in [8], [28], [33] (where the reported results in [28], [33] are based on four of the five projects except the Closure Compiler project, and the reported result in [8] is based on all the five projects). Our results (not in parentheses) are based on all the five projects.
† The results reported in [8] are based on a repair experiment on 90 selected bugs using a fault localization technique performed at the method level (with a faulty method known in advance). For each bug, the authors of [8] looked for a correct patch within the top 10 generated patches (if any). Our results are based on all the 357 bugs. The fault localization was performed at the project level. For a consistent comparison, we only checked the validity and correctness of the first generated patch (if any).
‡ In our experimental setup, we found that ACS (available at [31]) took longer than what is reported in the paper [28] to produce a plausible patch, and we did not reproduce many correct patches reported in [28].

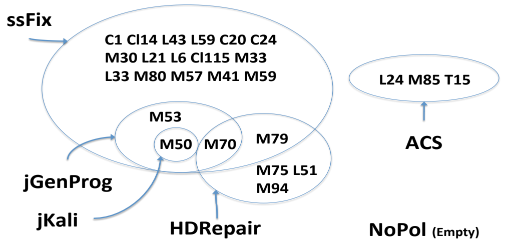

Fig. 3. Valid Patches Generated by Different Techniques

patches (using either less or comparable time) than these techniques did. All the valid patches generated by these techniques were actually generated by ssFix. jGenProg cannot practically produce a correct patch when the repair statement does not exist in the faulty program. It deletes a statement in high probability and this often leads to a defective patch generated. jKali can only delete a statement, so it is not expected to produce any correct patch that does not involve statement deletion. Nopol uses a conditional synthesis technique to produce patches related to an if-statement. Our results show that it is prone to produce a patch with a synthesized condition being too constrained or too loose. An example (M85) is shown below.

```
if (fa * fb >= 0.0) { ... } //The faulty statement
if (fa * fb > 0.0) { ... } //The correct patch
if (fa * fb >= 1) { ... } //Nopol's patch
```

Although Nopol created a patch by constraining the original if-condition to make the test suite pass, it is overly constrained and would not be correct in general.

HDRepair is an extension of GenProg. It uses more modification operations than GenProg does but leverages the bug-fix patterns mined from existing bug-fixing instances to make the patch search process guided. However, our results show that HDRepair's bug-fix-pattern driven algorithm is not truly effective. Compared to jGenProg, it only produced two more valid patches with the median repairing time being longer. Overall, ssFix outperforms HDRepair. But HDRepair did produce three valid patches that ssFix failed to produce. For example, to produce one of such patches (for M75), HDRepair reused a statement *return getPct(Long.valueOf(v))* from the class of the faulty statement *return getCumPct((Comparable<?>) v);* and applied a modification to replace *getCumPct* with *getPct*. ssFix did not find the repair statement since its local context is not similar to that of the faulty statement.

ACS is a recently developed technique that also uses condition synthesis to repair a program. It leverages techniques of test case analysis, document analysis, dependency analysis, and predicate mining to produce an if-statement with a synthesized condition that is likely to be correct. Our results show that ACS generated valid patches for three bugs that none of the other techniques successfully repaired. M85 is an example that ACS successfully repaired by synthesizing a correct if-condition as *fa*fb>=0.0&&!(fa*fb==0.0)*: it first identified a target expression (*fa*fb*), then performed keyword search over the GitHub repositories to find relevant predicates and produced the expression *!(fa*fb==0.0)*, and it finally produced the correct condition by conjoining this expression with *fa*fb>=0.0*. Through using relevant expressions from GitHub, ACS synthesized a correct condition that is neither too constrained nor too loose. Although ACS produced three valid patches that no other techniques produced, our results show that ssFix still outperforms ACS in terms of the number of valid patches generated and the repairing time. Since ACS is designed to repair bugs related to if-conditions, it is not easy for ACS to produce a direct, valid patch for bugs like L59 for which ssFix produced a correct patch by replacing a method argument *strLen* with another *width*.

C. Discussion

In our experiments, we referred to the developer patch associated with a bug to manually determine the validity and correctness of a patch generated by a repair technique. There can be in general other ways to define the validity and correctness of a patch. For a fraction of plausible patches generated by ssFix and other techniques, we cannot easily determine their validity or correctness, but it is possible that some of the generated patches are valid and correct even though they are not syntactically equivalent or similar to the developer patches. Even so, we do not believe there can be a significant fraction of valid/correct patches among such plausible ones, and we released all the plausible patches at *https://github.com/qixin5/ssFix/tree/master/expt0/patch*. Though possibly biased, a manual evaluation method like ours is commonly used to evaluate the quality of patches generated by current automated repair techniques. The problem however can be mitigated through using a held-out test suite (to quantify overfitting) and/or other approaches that can identify overfitting patches (e.g., [34], [35]).

The repair performance of ssFix significantly depends on the performance of code search, and we think there is still room for ssFix's code search to be improved (e.g., through using the proposed solution we mentioned in the last part of Section III-A1 and/or using a larger code database). With a better code search, ssFix can be more efficient, and can produce more valid/correct patches and less overfitting patches. (To produce less overfitting patches, ssFix may also be combined with techniques like [34]–[38]).

V. Related Work

ssFix leverages existing code fragments to produce patches for bug repair. SearchRepair [15] and CodePhage [14] are two repair techniques that are built on a similar idea. The main difference between ssFix and the two techniques lies in how they perform code search to find code for bug repair. ssFix uses syntactic code search while the two techniques use semantic code search (based on symbolic execution and constraint-solving, and program execution respectively). ssFix is related to a branch of automated repair techniques [1]–[4], [6], [8], [39] that work by first defining a set of modification rules to have a search space of patches created and then using different ways to search in the space for patches that are likely to be correct. relifix [6] is one of the techniques that is related to ssFix in that it produces patches based on the changed statements between two programs. However, relifix only targets on repairing regression errors, and it does not do cross-project code search, code translation or component matching. relifix uses more modification operations than ssFix does to produce patches. It uses randomness for patch generation while ssFix does not. Another branch of repair techniques [24], [27], [28], [32], [40]–[42] leverage synthesis techniques to produce patches. SPR [5] is a staged repair technique combines using modification operations and using condition synthesis to generate patches. Prophet [10] is built upon SPR and uses a probabilistic model for patch ranking. ssFix is related to these techniques but does not use any synthesis techniques to produce patches. ssFix is also related to many repairing techniques using formal specifications (e.g., [43]), focusing on specific fixing tasks (e.g., [44]) and providing suggestions and feedbacks (e.g., [45], [46]).

Techniques like SYDIT [47], LASE [48], Refazer [49], and Genesis [50] can extract, synthesize, and infer code transformations for bug repair and for other purposes. Different from these techniques, ssFix does not learn any transformations but directly leverages existing code from a database to produce patches for bug repair.

The way ssFix produces patches can be thought of as creating a hybrid of two pieces of code (i.e., the target and the candidate chunks of code). ssFix is thus related to works that do code transfer (or transplantation) [14], [51]–[54].

VI. Conclusion

In this paper, we presented our automated repair technique ssFix which performs syntactic code search to find existing code from a code database that is syntax-related to the context of a bug and further leverages such code to produce patches for bug repair. Our experiments have demonstrated the effectiveness of ssFix in repairing real bugs. In the future, we will look at using different code search techniques on a larger code database for a potential performance enhancement.

References

[1] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012, pp. 3–13.

[2] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: models and first results," in *ASE*, 2013, pp. 356–366.

[3] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[4] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[5] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ESEC/FSE*, 2015, pp. 166–178.

[6] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *ICSE*, 2015, pp. 471–482.

[7] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, 2015, pp. 24–36.

[8] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[9] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ICSE*, 2016, pp. 702–713.

[10] ——, "Automatic patch generation by learning correct code," in *POPL*, 2016, pp. 298–312.

[11] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *ESEC/FSE*, 2014, pp. 306–317.

[12] S. Sumi, Y. Higo, K. Hotta, and S. Kusumoto, "Toward improving graftability on automated program repair," in *ICSME*, 2015, pp. 511–515.

[13] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *ICSM*, 2011, pp. 283–292.

[14] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI*, 2015, pp. 43–54.

[15] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *ASE*, 2015, pp. 295–306.

[16] W. Janjic, O. Hummel, M. Schumacher, and C. Atkinson, "An unabridged source code dataset for research in software reuse," in *MSR*, 2013, pp. 339–342.

[17] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: an eclipse plug-in for testing and debugging," in *ASE*, 2012, pp. 378–381.

[18] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *ESEC/FSE*, 2010, pp. 147–156.

[19] "Apache Lucene," https://lucene.apache.org.

[20] B. Fluri, M. Wursch, P. M., and G. H. C., "Change distilling: Tree differencing for fine-grained source code change extraction," *TSE*, pp. 725–743, 2007.

[21] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *TSE*, pp. 654–670, 2002.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE*, 2007, pp. 96–105.

[23] Lucene, "Lucene Practical Scoring Function." [Online]. Available: https://lucene.apache.org/core/4_6_0/core/org/apache/lucene/search/similarities/TFIDFSimilarity.html

[24] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015, pp. 448–458.

[25] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA*, 2014, pp. 437–440.

[26] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java," in *ISSTA, Demonstration Track*, 2016, pp. 441–444. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01321615/document

[27] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *TSE*, vol. 43, pp. 34–55, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01285008/document

[28] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017, pp. 416–426.

[29] "SpoonLabs Nopol," https://github.com/SpoonLabs/nopol.

[30] "HDRepair," https://github.com/xuanbachle/bugfixes.

[31] "ACS," https://github.com/Adobee/ACS.

[32] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016, pp. 691–701.

[33] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, "Automatic repair of real bugs: An experience report on the Defects4J dataset," Arxiv, Tech. Rep. 1505.07002, 2015. [Online]. Available: http://arxiv.org/pdf/1505.07002

[34] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *ISSTA*, 2017, pp. 226–236.

[35] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *ESEC/FSE*, 2017, pp. 831–841.

[36] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *ESEC/FSE*, 2016, pp. 727–738.

[37] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Test Case Generation for Program Repair: A Study of Feasibility and Effectiveness," Arxiv, Tech. Rep. 1703.00198, 2017. [Online]. Available: https://arxiv.org/pdf/1703.00198

[38] X. Liu, M. Zeng, Y. Xiong, L. Zhang, and G. Huang, "Identifying patch correctness in test-based automatic program repair," Arxiv, Tech. Rep. 1706.09120, 2017. [Online]. Available: https://arxiv.org/pdf/1706.09120

[39] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *TSE*, pp. 54–72, 2012.

[40] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[41] L. DAntoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *CAV*, 2016, pp. 383–401.

[42] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *ESEC/FSE*, 2017.

[43] Y. Wei, Y. Pei, C. A. Furia, S. L. S, S. Buchholz, M. B., and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*, 2010, pp. 61–72.

[44] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *ECOOP*, 2011, pp. 609–633.

[45] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated synthesis of repair hints," in *ICSE*, 2014, pp. 266–276.

[46] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *PLDI*, 2013, pp. 15–26.

[47] N. Meng, M. Kim, and K. S. Mckinley, "Systematic editing: generating program transformations from an example," in *PLDI*, 2011, pp. 329–342.

[48] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," in *ICSE*, 2013, pp. 502–511.

[49] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ICSE*, 2017, pp. 404–415.

[50] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017, pp. 727–739.

[51] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *ISSTA*, 2015, pp. 257–269.

[52] P. Amidon, E. Davis, S. Sidiroglou-Douskos, and M. Rinard, "Program fracture and recombination for efficient automatic code reuse," in *HPEC*, 2015, pp. 1–6.

[53] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *ESEC/FSE*, 2017, pp. 95–105.

[54] T. Zhang and M. Kim, "Automated transplantation and differential testing for clones," in *ICSE*, 2017, pp. 665–676.