Noname manuscript No.
(will be inserted by the editor)

Automatic Patch Generation with Context-based Change Application

Jindae Kim · Sunghun Kim

the date of receipt and acceptance should be inserted later

Abstract Automatic patch generation is often described as a search problem of patch candidate space, and it has two major issues: one is search space size, and the other is navigation. An effective patch generation technique should have a large search space with a high probability that patches for bugs are included, and it also needs to locate such patches effectively.

We introduce ConFix, an automatic patch generation technique using context-based change application. ConFix collects abstract AST changes from human-written patches with their AST contexts to provide abundant resources for patch generation. These collected changes are only applied to possible fix locations with the same contexts for patch generation. By considering changes with a matching context only, ConFix selects a necessary change for a possible fix location more effectively than considering all the collected changes. Also, ConFix filters out fix locations with no collected changes in the same context, which means that such locations have not been modified in human-written patches, hence they are not desirable for modifications.

We evaluated ConFix with 357 real bugs from Defects4j dataset. ConFix successfully fixed 22 bugs including six bugs which were not fixed by compared existing techniques. With context-based strategy, ConFix checked average 48% less fix locations than a strategy using only a spectrum-based fault localization technique until patches were generated. Also, it ranked changes required for patches at the top for 63.6%, and within top-3 for 81.8% of the fixed bugs.

Keywords automatic program repair · automatic patch generation · context-based change application

1 Introduction

Automatic Program Repair (APR) techniques have shown great progress recently. Many of these techniques are automatic patch generation techniques, which usu-

Jindae Kim · Sunghun Kim
The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
E-mail: jdkim@cse.ust.hk, hunkim@cse.ust.hk

ally generate a source code patch by modifying buggy code. One popular approach of automatic patch generation is the *generate-and-validate* approach used in many techniques [9, 15, 18, 23, 28, 30, 45–47, 56, 58, 60], which keeps generating and validating patch candidates until a patch - a candidate passes all given test cases - is found.

Automatic patch generation is often described as a search problem in a patch candidate space [28–30, 33]. In the candidate space of a technique, each point corresponds to a patch candidate which the technique can generate. Two key axes of this space represent where to fix (fix locations) and how to fix (applicable changes), which are defined by the technique's design. Then the patch generation is considered as an exploration of the space from a point to another point, by producing candidates until it reaches to a genuine patch.

The search problem has two major issues: one is search space size, and the other is navigation. A technique's search space may not contain a patch for a bug, if it only has limited pre-defined changes for patch generation [9,18,23,28,30,47,54,56], or it modifies only certain types of fix locations [9,18,28,30]. To expand the search space and generate various patch candidates, recent techniques mine changes from human-written patches [15, 20, 27, 33, 58] or collect code fragments from software repositories to provide resource for patches [17, 52, 60].

However, such expansion makes the navigation for a patch even more difficult, since actual patches which can fix a bug are very sparse in a huge search space [29]. To address this navigation issue, automatic patch generation techniques employ various strategies such as using natural distribution of changes [15, 33, 58], integer linear programming [27], or leveraging syntactic and semantic characteristics of code [14, 17, 21, 60].

To address both issues in the search problem, we introduce a novel automatic patch generation technique *ConFix*, which uses Context-based Change Application technique (CCA) to generate patch candidates. The key idea of ConFix is that it collects abstract changes as well as their Abstract Syntax Tree (AST) contexts, and applies a collected change to a possible fix location only if their AST contexts are matched. For example, consider a change which inserts a null checker before a statement.

```
  n = t.getNode(label);
+ if(n != null) //Example 1
    p = n.getParent();
+ if(n != null) //Example 2
    Node v = new Node();
```

In the above code snippet, `n` might be null if `t.getNode(label)` returns null. Then `n.getParent()` will cause *NullPointException* (NPE), but inserting a null checker before it (Example 1) will fix the problem. However, for Example 2, the variable declaration will not cause NPE, which indicates that the null checker will be useless. Even worse, it may introduce a new error since the change makes variable `v` no longer available for the following statements.

Note that these two changes are identical null checker insertions. Without consideration of contexts, an automated technique probably will apply this change anywhere and produce many patch candidates like the second case. On the contrary, it is unlikely that human developers make changes like the second example, since it is error-prone and not meaningful to solve the NPE issue.

ConFix uses AST contexts to avoid the second example and to produce more candidates like the first example. ConFix compares AST contexts of a change and a target location defined by parent, left and right nodes, and applies the change to the location only if their contexts are matched. In ASTs, these nodes represent nearby code fragments, hence ConFix can verify whether code fragments around a target location is similar to nearby code fragments when a change was collected.

Context-based strategy provides two advantages to ConFix for search space navigation. Firstly, ConFix can avoid changes which developers may not produce, and tries to mimic collected human-written changes more similarly. For each possible fix location, ConFix only considers changes which have been applied to similar locations by developers, and it will reduce the search area by filtering out the other undesirable changes. Secondly, ConFix can also avoid to select fix locations which developers may not want to modify. If a possible fix location has no collected changes with the same context, it means that developers have not modified such locations to fix bugs in collected patches. ConFix's search area can be even more reduced by filtering out such unlikely fix locations. Therefore, ConFix can expand its search space by collecting more changes, while it can navigate through them effectively with the guidance of contexts.

We evaluated ConFix with 357 real Java bugs from Defects4j dataset [16]. To provide changes for patch generation, we collected over 6K human-written patches from nine different projects and identified about 216K abstract AST changes (44K unique changes). We also identified 38K (more specific) and 2.4K (more abstract) contexts for the collected changes. ConFix exploited these changes and contexts to generate patch candidates until it found a patch for each bug. When using more specific AST contexts, ConFix only needed to consider average two changes per each possible fix location, instead of selecting one of the 44K unique changes.

We manually inspected generated patches to assess correctness, and compared them with patches of existing techniques [15, 58, 60]. ConFix correctly fixed 22 bugs which is comparable to 18-26 bugs of the compared techniques. Also, 6 out of 22 fixed bugs are fixed only by ConFix. We also found that ConFix fixed about 74% of all bugs whose patches - necessary changes - were included in the collected changes. This result indicates that ConFix might fix even more bugs if we collected more changes for patch generation.

We also analyzed the effectiveness of ConFix's strategies. With context-based strategy, ConFix checked average 48% less fix locations until it fixed bugs, compared to a strategy which used a spectrum-based fault localization only. By filtering applicable changes with contexts, ConFix also ranked changes applied to fix bugs within top-1 for 63.6%, and top-3 for 81.8% of the fixed bugs.

The followings are major contributions of this study.

- We introduce ConFix, an automatic patch generation technique which leverages human-written changes and their contexts for patch generation.
- We provide empirical evaluation results of ConFix and generated patches on Defects4j benchmark dataset.
- We present a detailed analysis on the effectiveness of ConFix's strategy to leverage collected contexts and changes for patch generation.

Remainder of this paper is organized as follows. First we introduce our context-based change application technique in Section 2. Then we elaborate how ConFix generates patches with the change application technique in Section 3. We provide

experimental setup for evaluation in Section 4 and analyze its results in Section 5. After that, we discuss about threats to validity of this study (Section 6) and related work (Section 7). Finally we conclude with future work in Section 8.

2 Context-based Change Application Technique

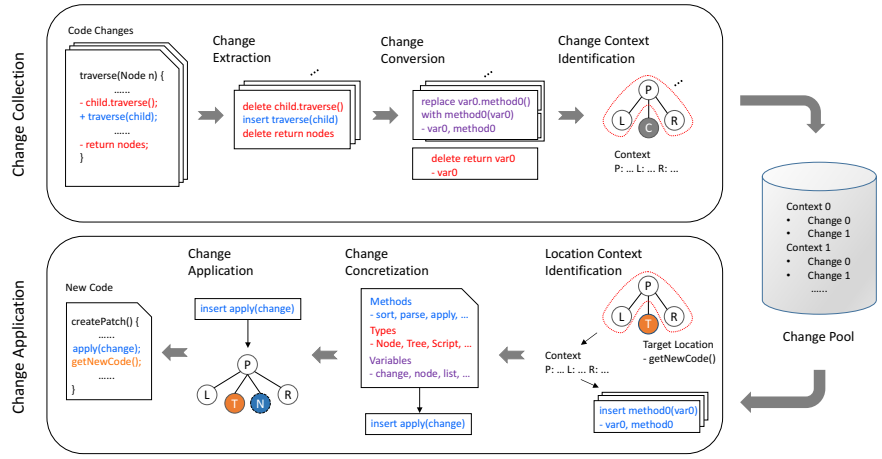


Fig. 1: The Overview of Context-based Change Application Technique

Fig. 1 shows the overview of the context-based change application technique. Our context-based change application technique consists of change collection and application phases. In change collection phase, the technique mines abstract AST subtree changes as well as their AST contexts from human-written patches to build a change pool. In change application phase, a target location should be given for modification. Then it retrieves changes with the same AST context as the location's context, and one of the changes is selected and applied to the target location.

2.1 Change Collection

The main purpose of change collection is building a change pool by collecting changes which are generally applicable to other locations. However, changes represented by source code form (e.g., unified diff format) are not suitable for this purpose, since they are easily affected by specific user-defined identifiers or coding style. To avoid this issue, we extract AST subtree changes from human-written patches by applying a source code differencing technique (*Change Extraction*). Then the extracted changes are further polished to more generally applicable form by several steps of conversion (*Change Conversion*). Finally, the changes are categorized by their AST contexts and stored in a change pool (*Change Context Identification*).

Change collection is the key step of the technique, since change application will be highly dependent to how changes and their contexts are collected. A change pool is the final outcome of change collection phase, and it will be used repeatedly as a repository of fix ingredients during change application phase.

2.1.1 Change Extraction

```
 public String getHashString(Node node){
  StringBuffer sb = new StringBuffer();
  //Add parent node.
- sb.append(node.parent);
+ if(node.parent != null)
+   sb.append(node.parent);
  //Add node label.
- sb.append(node.type);
+ sb.append(node.label);
  //Add child node hash.
  for(Node child : node.children){
    sb.append(getHashString(child));
  }
- sb.append("\n");
  return sb.toString();
 }
```

(a) An Example Source Code Patch

```
1:insert(IfStatement, Block)
    insert(InfixExpression:!=, IfStatement)
      insert(QName:node.parent, InfixExpression:!=)
      insert(NullLiteral, InfixExpression:!=)
2:move MethodInvocation from Block to IfStatement
3:update(QName:node.type, QName:node.label)
4:delete(MethodInvocation, Block)
    delete(SimpleName:sb, MethodInvocation)
    delete(SimpleName:append, MethodInvocation)
    delete(StringLiteral:"\n", MethodInvocation)
```

(b) An Edit Script with Four Edit Operations

Fig. 2: An example source code patch and an edit script generated from the patch. In the edit script, indentations indicate the hierarchy of the changed AST subtree. Note that the move operation only shows the root node for simplicity.

Fig. 2a is an example source code patch represented in unified diff format. There are three change hunks in the example. The first hunk represents an insertion of a null checker, the second one shows that a field `type` is updated to `label`, and the last one is a method call deletion. During change collection step, each change hunk will be extracted and converted to an individual AST subtree change.

Note that one patch may consist of several individual changes applied to multiple locations. We collect separate individual changes rather than the entire patch as one single change. Since we are leveraging the repetitiveness of changes, using small changes increases the probability that such changes are repeatedly used to generate other patches [41]. Also, dividing a patch into small individual changes and composing them for another patch is proven to be effective [33, 64].

To extract individual changes from two versions of source code (i.e., one source code change), we used a source code differencing tool [19] which generates AST subtree edit operations. First, it matches AST nodes between old and new ASTs to identify unchanged nodes. Then unmatched nodes in the old AST indicate that they are deleted, and unmatched nodes in the new AST means they are inserted. Hence delete and insert operations are generated for those nodes. If there exist matched nodes which have different positions in the two ASTs, they are considered moved, and move operations are generated for them. Update operations represent matched nodes with different node values such as the second hunk of Fig. 2a, which updates a variable name. Although we employed a specific tool for implementation of our technique, any source code differencing technique works on ASTs (e.g., GumTree [10]) can be adapted for our technique.

Fig. 2b shows a string representation of an edit script generated from the example patch in Fig. 2a. Each line represents a node edit operation, consists of change type and label of a changed node and its parent. Each node label has the form of *node type:node value*. Node type is AST node type, and node value indicates specific identifiers, literals or operators represented by a node. An indented line means that the line (i.e., an edit operation) is a child of a previous less indented line. Hence lines with the same type of edit operations form a tree to represent a subtree edit operation.

2.1.2 Change Conversion

Source code differencing techniques are originally designed to describe changes with pre-defined edit operations. However, our purpose of change extraction is to obtain changes applicable to other locations to reproduce similar changes. Generated edit operations are often not suitable for our purpose, hence they need to be polished further.

For instance, the first hunk in Fig. 2a is represented by the first two edit operations in Fig. 2b. The insert operation only represents the first added line of the hunk. The deleted and added method calls (`sb.append(node.parent)`) in the first hunk are considered moved since they are identical, and represented by the move operation. These insert and move operations are inseparable and represent one single change. If the insert operation is applied to somewhere else independently, it may cause errors since the added code fragment is not a complete if statement. Also, the move operation itself is more meaningful when it is combined with the inserted if statement.

To address this issue, we employ *replace* change type, and convert such edit operations into one replace type change. We can express the insert and move operations with one replacement which replaces the method call with the if statement. Replace type changes can be also used to combine delete and insert pairs happened at the same location.

The change conversion is done by the following steps.

1. Divide move operations into a pair of delete and insert operations.
2. Extend each insert or delete operation to all descendant nodes of its changed nodes.
3. Remove any operation if a changed AST is a subtree of another operation's changed AST.

4. Find all insert-delete pairs applied to the same location, and combine them into one replace type change.

Change conversion basically combines insert-delete pairs whose changed subtrees are overlapped. For example, the insert operation in Fig. 2b consists of four AST nodes, which only indicate the first added line of the first hunk. The moved method call is divided to delete and insert operations by Step 1, and the inserted method call is overlapped after Step 2, since it belongs to the inserted if statement. By Step 3, the method call insertion is removed, and the remaining method call deletion is combined with the if statement insertion at Step 4 to generate a replace type change.

```
1:replace
sb.append(node.parent)
with
if(node.parent != null)
    sb.append(node.parent)
2:update
node.type
to
node.label
3:delete
sb.append("\n")
```

(a) Combined Changes

```
1:replace
var0.append(var1)
with
if(var0 != null)
    var1.append(var0)
2:update
var0
to
var0
3:delete
var0.append(str0)
```

(b) Normalized Changes

Fig. 3: Combined and Normalized Changes obtained from Edit Operations during Change Conversion

Fig. 3a shows the string representation of changes after overlapped edit operations are combined. For each change, we present related code fragments instead of AST subtrees to show changed parts in source code more clearly. As we explained, the insert and move operations in Fig. 2b are combined to one replace type change in Fig. 3a.

In CCA for ConFix, we discarded all delete and move operations after edit operations are combined. Recent study showed that using delete and move operations often produces incorrect patches and not helpful to patch generation [15,55]. During change conversion, some of the delete and move operations are already combined to more generally applicable replace changes. Hence we decided to ignore remaining delete and move operations which may not be helpful. Note that we show a delete change in Fig. 3 only for an example, although it is not actually collected.

After edit operations are combined, we have three types of AST subtree changes.

Definition 1 (AST Subtree Changes)

- **insert** t: insert a subtree t.
- **update** v_1 to v_2: update a node's value from v_1 to v_2.
- **replace** t_1 with t_2: replace a subtree t_1 with t_2.

Combined changes are still not ready for general application, since they contain many concrete values such as variable, type, method names and string literals. These values are usually dependent to codebase where changes are extracted, and not available at different locations. By normalizing changes, we can adapt these

changes to their applied locations with variables, types, and methods available at the locations. String literals are also normalized, since they are often log messages which are not useful at other locations. Also, if they are not normalized, identical changes with different string literals will be considered different changes, which may lead to ineffective dispersion of changes.

The technique normalizes changes with two principles: consistency and order preserving. Consistency means that the same identifier or string literal should be replaced with the same unique abstract name. For example, if variable `node` is used twice in an AST subtree, the two occurrences must be replaced with the same abstract name such as `var0`. If there is another variable `sb` is used in the same subtree, it should be replaced with another abstract name like `var1`, other than `var0`. Order preserving indicates that the technique should assign the same abstract name to user-defined names or string literals shown in the same location of two identical subtrees. For instance, variables `sb` and `strBuf` in two inserted code fragments `sb.append()` and `strBuf.append()` should be replaced with the same abstract name `var0`. In this way, the technique groups changes with the same changed code structure into one single change.

During normalization, additional information about normalized names are collected to support change concretization. This information is mostly related to types, such as variable types, the signature of methods - which consists of return type and parameter types. At the time of change concretization, the collected information can work as requirements for each abstract name which ConFix should consider when it assigns a concrete name for the abstract name.

Fig. 3b shows changes normalized from the combined changes in Fig. 3a. Note that old and new subtrees related to a change are normalized separately. For instance, in replace change, variable `sb` is normalized to `var0` and `var1` in old and new subtrees respectively, since the two subtrees were separately normalized.

These subtrees are normalized separately since the purpose of normalization is different. During change application, only abstract values on a new subtree are replaced with concrete values. Hence CCA normalizes the new subtree and collects necessary information. On the other hand, an old subtree is normalized to identify the same changes and aggregate them. Two changes are considered identical if their change types are equal and their corresponding subtrees are label-isomorphic. Since a node's label contains specific names, they should be normalized to aggregate changes with the same changed subtrees regardless of user-defined names used in them.

In case of replace type changes, abstract names are not consistent in both subtrees with this method, but we chose flexibility over strict reproduction. When a replace type change is applied, original concrete names of abstract names appeared on both subtrees will be fixed if the abstract names are consistent in both subtrees. This leads to the strict reproduction of collected replace changes. However, our main purpose is providing various changes to generate various patches. If a new subtree is normalized separately, one abstract replace change can produce various concrete replacements by assigning different concrete names regardless of abstract names in an old subtree. Therefore, separate normalization is more suitable for our purpose.

2.1.3 Change Context Identification

After individual changes are obtained from source code patches, the next step is identifying AST contexts of the changes. A context of a change describes code fragments near the change, hence we can apply changes to target locations with similar neighbor code fragments by comparing contexts. In this way, we can avoid meaningless or erroneous modifications such as inserting a statement after a return statement, which will turn into unreachable code.

To define AST context, we use neighbor nodes of a changed AST subtree. More precisely, we use the parent of the subtree's root, the root's left and right nodes to define contexts. Since both change collection and application happen in AST level, it is more convenient to define contexts with AST nodes. In source code level, these nodes represent code fragments near a change. Parent node represents the location where a changed code fragment belongs. Left and right nodes indicate code fragments before and after the changed code fragment respectively.

Note that we will use contexts to verify that code fragments near a target location is similar to code fragments near a change. To achieve this goal, we employ source code finger-printing techniques [6] which often used to identify similar code. More specifically, we use the node type and node type hash of nodes to define the AST context. The node type is the AST node type of an AST node, which simply represents the type of code fragments (e.g., assignment, method call). The node type hash is more specific representation which can represent the structure of the whole AST subtree rooted at an AST node.

We use Dyck word hashing [6] to obtain node type hash value.

Definition 2 (Dyck Word Hash) Dyck word D_n of node n with node type $n.type$ is defined as follows.

$$D_n = \{n.type\{D_{c_1}, D_{c_2}, \cdots, D_{c_k}\}\}, \text{where } c_1, c_2, ..., c_k \text{ are child nodes of } n. \quad (1)$$

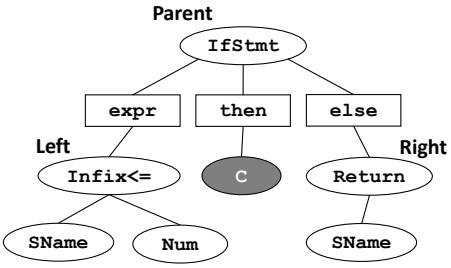
A hash value contains both type and structure of AST nodes which correspond to a code fragment. Parent-child relation is represented by surrounding each node's child nodes with brackets, and each node is represented by its node type. Expressions with operators also include specific operators (e.g., +, -, ==, etc.) in its node type. By comparing two nodes' hash values, we can check whether they are the roots of two type-isomorphic AST subtrees. Hence contexts using hashes provide tighter constraints than using node type.

There are various possible combinations of nearby nodes with their node type or node type hash for contexts. Among them, we use two specific combinations, **P**arent **T**ype, **L**eft and **R**ight **H**ash (***PTLRH***) and **P**arent, **L**eft, **R**ight **T**ype (***PLRT***) contexts in this study. PTLRH and PLRT contexts are the simplest form of contexts using all three nodes (parent, left, right) with node type hash and node types, and they showed promising performance in a preliminary experiment with a few hundreds changes to find a right change for a location. Context types more simpler than PTLRH and PLRT cannot distinguish contexts enough, hence there are too many applicable changes for each context. On the other hand, context types which are more specific than them scatter changes excessively, hence it is difficult to find matched context in other locations. Therefore, we decided that PTLRH and PLRT contexts are suitable for ConFix.

Definition 3 (PTLRH and PLRT Contexts) For a given change *change* and its changed subtree's root c, let nodes p, l, r be the parent, left and right nodes of c respectively. Then PTLRH and PLRT contexts of *change* are strings of the following form.

- ***PTLRH*** = P:$p.type[c.loc]$, L:$l.hash$, R:$r.hash$
- ***PLRT*** = P:$p.type[c.loc]$, L:$l.type[l.loc]$, R:$r.type[r.loc]$

where $n.type$ indicates the AST node type, $n.loc$ represents syntactic location, and $n.hash$ denotes Dyck word hash of n.



(a) An AST representing The Change

```
PTLRH Context
P:IfStmt[then],
L:{Infix<={SName,Num}},
R:{Return{SName}}

PLRT Context
P:IfStmt[then],
L:Infix<=[expr],
R:Return[else]
```

(b) An Example Change

Fig. 4: PTLRH and PLRT context identified from a code change. Node C is the root of a changed AST subtree. Rectangular nodes represent syntactic locations.

Fig. 4a shows PTLRH and PLRT contexts of an example AST which represents the following code change.

```
if (count <= 10) {
+ count++;
}else{
  return sum;
}
```

Node C is the root of the changed AST subtree representing `count++`, `IfStmt` node is the parent node, `Infix<=` is the left node and `Return` is the right node. Rectangular nodes are virtual nodes to show nodes' syntactic location. For example, the inserted node is under the if statement's then-block, and the return statement belongs to the else-block. Without the virtual nodes, we cannot distinguish each node belongs to which block.

PTLRH context shows the type and syntactic location of the parent node and the hash of left and right nodes. The technique actually uses the syntactic location of node C (`then`) as the parent node's syntactic location $p.loc$, since the changed subtree's syntactic location is the one we are actually interested, not where the parent node belongs. In PLRT context, syntactic locations of left and right nodes are their own syntactic locations. If any of the parent, left or right nodes is missing, the technique simply ignores missing nodes and uses the rest for the context.

When we match contexts of a change and a target location, both PTLRH and PLRT contexts ensure that the change and location are under the same kind of code fragment. The difference between PTLRH and PLRT contexts is that PTLRH contexts distinguish code fragments, while PLRT contexts can only recognize the

kind of code fragments. For example, consider the following code change and its application at another location:

```
//Original Change
  Type var = new Type();
+ var.method();

//PTLRH: prevented, PLRT: allowed
  Type var = method0();
+ var.method();
```

When PTLRH context is being used, applying the change after `Type var = method0()` is not allowed, since it is actually a different code fragment from the original change's context. As a result, PTLRH context prevents a change which may cause an error if `method0()` returns null. If we use PLRT context, the original and new changes are both after a variable declaration, hence the new change is allowed. We still need PLRT context, since many changes can be applied without any errors although nearby code fragments are slightly different.

One exceptional case in context identification is `Block` nodes. Normally, a `Block` node simply groups statements and does not specify the type of code block. Therefore, using a `Block` node as the parent node of the changed AST subtree cannot provide enough information about the context. To avoid this issue, we use a `Block` node's parent to represent the context. In this way, we can distinguish whether a change is applied to a block under a while loop or an if statement.

Once a change's context is identified, the change is stored in a change pool under the identified context. We can retrieve this change from the change pool when we need to apply a change at the same context. Note that we have two types of change pools categorizing changes with PTLRH and PLRT contexts respectively. After contexts are identified and change pools are built, change collection step is finished and changes are ready for application.

2.2 Change Application

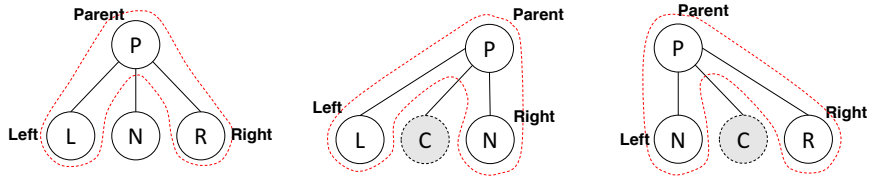
In this section, we explain how the technique applies a change to a given target location. Basically, it first identifies the target location's context. Then changes with the same context are retrieved from a change pool. One of these changes are selected (*change selection*) and all normalized values are replaced with concrete values (*change concretization*) available at the target location. After that, the concretized change is applied to the target location.

2.2.1 Target Location Context Identification

The first step of change application is identifying the context of a given target location. Since all changes operate on ASTs, a target location is also an AST node which a change will be applied. Hence our method of location context identification is similar to change context identification.

The most important requirement of location contexts is that they should be compatible with change contexts. If a target location has the same parent, left and right nodes as a changed AST subtree, contexts of the location and change must be identical. For update and replace type changes, there exists a changed

AST subtree in an old AST before the change is applied. We can simply identify location contexts for such changes just like we identify change contexts, by using parent, left and right nodes of a target location. However, the context of an insert change only describes the situation after the change is applied. Hence we need to assume cases a new subtree is inserted before or after a target location to match the location context with contexts of insertions.



(a) Default **(b) Insert Before** **(c) Insert After**

Fig. 5: Default, Insert Before and Insert After Target Location Contexts. Node N indicates a target location which contexts are identified. Node C represents a placeholder for Insert Before and Insert After contexts where a change will be actually applied.

Fig. 5a shows how we identify *Default* context, which is identical to change context. Node N is a target location, and nodes P, L, R represent parent, left and right nodes of node N respectively. Then we can use these nodes to identify Default context of the target location defined by Definition 3. *Insert Before* and *Insert After* contexts are the contexts for cases which a new AST subtree is inserted before and after node N respectively. In Fig. 5b and 5c, node C indicates the actual location where a subtree will be inserted. Then node N is considered as right node (Insert Before) and left node (Insert After) of contexts, to be matched with the inserted subtree's right and left nodes respectively.

2.2.2 Change Selection and Concretization

Next step is *Change Selection*, which literally means that one of the collected changes is selected for application. In a change pool, changes are categorized by their contexts and their frequencies are also stored. Hence CCA can easily retrieves a list of changes with the same context as the target location in descending order of their frequencies.

Various strategies for change selection can be used to select a change from the retrieved list. However, all these strategies should check whether the target location is actually matched to the old changed subtree of changes except for insert type changes. For instance, when `replace var0 with var0.method0()` is applied to a method argument, matching context does not guarantee that the target location (i.e., the method argument) is actually a variable `min` or a method call `str.length()`. For the latter, ConFix cannot apply the replacement, hence such changes should be removed from the list. Filtering can be done by comparing node type hash of a target location and a change's old subtree. ConFix's change selection for patch generation will be explained in Section 3.4.

After a change is selected, CCA needs to replace all normalized values of the change with concrete values. We call this step as *Change Concretization*, which adapts changes to given target locations. Similar to change selection, CCA does

not specify a certain change concretization strategy, but it only provides informations related to normalized values obtained during collection. ConFix's strategy for change concretization will be explained in Section 3.3.

Once the change is concretized, the last step is applying the change to the target location. Change application itself is straightforward since it simply inserts or replaces a subtree or updates a value of a node. After change application is finished, we have a modified AST which can be reverted to modified source code.

3 Patch Generation

3.1 Patch Generation Process

ConFix is a generate-and-validate technique, which keeps generating patch candidates one by one until one of the candidates passes all given test cases. To generate a patch candidate with CCA, ConFix first needs to identify fix location candidates, and select one of them. Once a fix location is selected, ConFix also selects one of the applicable changes retrieved from a change pool. Then ConFix concretizes the selected change for the target location and applies the change to generate a patch candidate. Finally, the generated candidate is compiled and verified by given test cases.

The actual patch candidate generation with a change pool is done by the following process.

1. ConFix identifies all code lines covered by given test cases.
2. For each covered line, ConFix identifies fix location candidates from the lines.
3. For each fix location candidate, ConFix tries to apply a certain number (*maxChangeCount*) of applicable changes.
4. For each change, ConFix tries to concretize them for *maxTrials* times.

Such candidate generation continues until one of the following termination criteria is satisfied.

- Current patch candidate passed all given test cases.
- ConFix generated and validated a certain number (*maxCandidates*) of patch candidates.
- ConFix consumed all time budget.

The first case means that ConFix successfully generated a patch. The next two conditions limit time and resources, and ConFix considers patch generation is unsuccessful if it cannot find a candidate passing all test cases.

3.2 Identifying Fix Location Candidates

To identify fix location candidates, ConFix needs to identify and order lines covered by given test cases. First, ConFix uses Ochiai [40], a spectrum-based fault localization (SBFL) technique to compute each code line's suspiciousness score ranges from 0 to 1 - close to 1 is more suspicious. Then lines with score greater than 0 are sorted on descending order of their scores.

For each covered line, ConFix identifies fix location candidates in the code line. ConFix first lists up all AST nodes correspond to the covered line and identifies

their contexts (Section 2.2.1). Note that for each AST node, CCA finds three different types of contexts. Then ConFix filters out contexts with no applicable changes in a change pool. Each of the remaining contexts can represent a fix location candidate with applicable changes.

After fix location candidates are identified, ConFix uses these candidates and their frequencies to sort covered lines with the same score.

Definition 4 (Context and Line Frequency) Let $F(c)$ be the frequency of a change c. Then the frequency of a context x can be defined as follows.

$$F_C(x) = \sum_{c \in C} F(c), \text{where } C \text{ is a set of applicable changes under context } x. \quad (2)$$

A covered line l's frequency $F_L(l)$ can be defined with F_C.

$$F_L(l) = \sum_{x \in L} F_C(x), \text{where } L \text{ is a set of all contexts appeared in line } l. \quad (3)$$

A covered line with higher frequency means that this line has been more frequently modified by past patches. ConFix uses this information to sort the same score lines in descending order of their frequency, to break ties in the score. In this way, ConFix tries to modify covered lines which have been modified more frequently in the past.

Fix location candidates in each line is sorted by ascending order of their number of applicable changes. Each covered line usually contains only several fix location candidates. However, some of these fix locations have a lot of various applicable changes including low frequency changes, and result in generation of many patch candidates and a waste of time budget. This is not desirable, since if applying some changes to the current location did not fix a bug, it is more probable that the rest of the changes also cannot fix the bug. We found that ordering fix locations with their frequency cannot prevent such bottleneck. Therefore, ConFix checks fix location candidates with less changes first to avoid bottleneck and increase the number of completely checked fix locations.

However, prioritizing candidates still cannot prevent that ConFix stays on a certain covered line, since all fix location candidates on the line will be checked eventually. Hence we employed *maxChangeCount*, to limit the number of changes applied for each location candidate and investigate more fix locations within limited resources and time.

3.3 Concretization Strategy

To use CCA for patch candidate generation, ConFix needs to specify how to convert abstract changes to concrete changes. Change concretization can be done by assigning concrete values to abstract values in a change. ConFix employs *Hash Match* and *Type Compatible (TC)* methods, to collect and assign concrete values.

3.3.1 Hash Match Method

Hash match method first collects code fragments and their node type hash (Definition 2) from buggy code. When concretizing a change, hash match method lists up code fragments with the same node type hash for a changed subtree of the change. Then it replaces all abstract values with concrete values of corresponding AST nodes representing one of the code fragments. With this method, ConFix can borrow code fragments closely related to buggy code as human-written usage examples.

Consider a situation which ConFix tries to concretize a change `replace var0==0 with var0>0` for buggy code with two code fragments as follows.

```
strLen == 0 // buggy code
var0 > 0  // a replacement

index > -1  // fragment 1
strLen > 0  // fragment 2
```

For change concretization, ConFix needs to assign a concrete variable name to `var0` in `var0>0`. Instead of finding a concrete variable alone, it tries to find code fragments with the same structure by comparing node type hashes. Suppose that ConFix found two code fragments with the same node type hash {`Infix{SName,Num}`} as shown in the example. Note that node type hash for change concretization does not include specific operators as shown in Fig. 4a. If we considered operators and literals in type hash, changes might not have any matched code fragments, hence we added a little flexibility in borrowing code fragments.

After code fragments are listed, ConFix selects a code fragment most closely related to the buggy code. To compute closeness of the buggy code and a code fragment, ConFix breaks all identifiers appeared in both of them into tokens, and creates token vectors containing token frequencies. Buggy code and fragment 1 have three tokens `(str, Len, index)`, and token vectors are $(1, 1, 0)$ and $(0, 0, 1)$ respectively. Similarly, for buggy code and fragment 2, token vectors are both `(str, Len)`$=(1, 1)$. Then ConFix simply computes the distance between token vectors, and selects fragment 2, since it is more close to the buggy code. Note that in case of the same distance, ConFix tries more frequent code fragment first.

However, this method might not work with single node changes such as update type changes or inserting a variable. Since their changed subtrees only have one node, any node with the same node type will be listed. Hence human-written code fragments are less useful as usage examples.

To address this issue, ConFix estimates a code fragment after a single node change is applied, and tries to find code fragments which can be used for the estimated code fragment. Consider the following buggy code from Defects4j:

```
return solve(min, max); // buggy code
insert var0 // a single node change
solve(var0, min, max) // an estimated code fragment
```

In this example, the single node change is an insertion of a variable. Suppose that ConFix inserts `var0` before variable `min`. The estimated code fragment corresponds to the inserted variable's parent node and its subtree. Since `var0` is inserted as a method argument, its parent is a method call in the example, and the estimated code fragment after the insertion is `solve(var0, min, max)`. To decide

`var0`'s concrete value, ConFix lists up all code fragments same as the estimated code fragment except for a variable at `var0`'s location (e.g., `solve(f, min, max)` or `solve(g, min, max)`). ConFix selects one of them and replaces `var0` with a concrete variable at the same location. For instance, if `solve(f, min, max)` is selected, `var0` will be replaced with variable `f`.

Note that all listed code fragments have the same token vector distance, since they are only different for one single node. ConFix selects one of the most frequently appeared code fragments. Frequent code fragments indicate that they are commonly used in buggy code, hence it is safer to use them in other locations.

ConFix treats update changes similarly. If variable `min` is updated in the example buggy code, ConFix finds code fragments of the form `solve(var0, max)`, and assigns the concrete variable appeared at `var0`'s position in the most frequent code fragment.

ConFix uses two sets of code fragments collected from two different scopes. One is code fragments from the buggy class which the current fix location belongs (*hash-class*). The other is code fragments from the other classes in the same package of the buggy class (*hash-package*). Code fragments from the same buggy class is more likely to be related to fix locations and they will cause less compilation errors than some random code fragments. The other classes in the same package might be helpful since they often implement similar functionalities, and some classes inherited from the same class may share code fragments. ConFix first checks the buggy class, and if there is no usable code fragments, it expands the search to the same package classes.

3.3.2 Type Compatible Method

TC method collects identifiers from buggy code, and selects a concrete name for each abstract name by considering type compatibility. Suppose that ConFix applies `insert method0(var0)`. There are constraints for concrete methods and variables to replace `method0` and `var0` without violating type compatibility. For instance, a concrete method must have one parameter, and the parameter type should be compatible to the type of a concrete variable assigned to `var0`. In addition, if this method call is inserted to another method as a method argument, the method's return type needs to be compatible to the expected parameter type.

In TC method, ConFix first assigns concrete methods to abstract methods. Assigning a concrete method to an abstract method usually needs to consider more constraints due to various types (e.g., return type, parameter types) used in methods. Hence ConFix replaces abstract methods with compatible concrete methods first, then it selects concrete types and variables according to fixed concrete methods. Since all user-defined types are normalized, ConFix uses the abstract signature of a method to check method compatibility.

Definition 5 (Abstract Signature) $AbsSig_m$, the abstract signature of a method m is a string defined as follows.

$$AbsSig_m = T_{decl} :: T_{ret} :: [T_p^1, T_p^2, ..., T_p^n] \tag{4}$$

where T_{decl} is a normalized type which m is declared, T_{ret} is the return type of m, and T_p^i, $i = 1, ..., n$ are the parameter types of m. Note that only user-defined types are normalized.

Definition 5 defines the abstract signature of a method. Basically, the abstract signature is a method signature with normalized types. Types appeared in a method signature are normalized in the same way we normalized a change described in Section 2.1.2. For example, suppose that we are trying to obtain the abstract signature of the method call `getHashString(child)` shown in Fig. 2a. The concrete signature of the method `getHashString()` is `TreeUtils::String::Node`, where `TreeUtils` is the class which the method is declared, and `Node` is the user-defined class of its parameter. In the abstract signature, user-defined types are normalized to `Type0::String::Type1`. If we find a concrete method with the same abstract signature, then at least it will not cause compile errors and can be replaced with the normalized method.

Definition 6 (Assignable Types) Type T_1 is assignable to type T_2, if $|V_1| \geq |V_2|$ ($|V_i|$ is the number of available variables of T_i), and T_1, T_2 satisfy one of the following conditions.

- Both T_1 and T_2 are normalized types.
- T_1 is a type from Java Standard Library(JSL) and T_2 is normalized type.
- Both T_1 and T_2 are from JSL and they are compatible.

Definition 6 shows the conditions to assess whether T_1 is assignable to T_2. ConFix considers both type compatibility and the number of variables of the types. Assessing type compatibility of two concrete types are simple and identical to typical judgment, such as a subtype is compatible to its super type, and primitive types (e.g., `int, double`, etc.) are mutually compatible to their wrapper types (e.g., `Integer, Double`, etc.). Normalized types are considered as wildcard characters, which means that they can be assignable to either normalized types or JSL types.

ConFix considers the number of variables in each type to prevent the situation that variable concretization is blocked by the type assignment. For example, suppose that ConFix assigns a concrete type T_1 to a normalized type T_2 to concretize a change. If there are three normalized variables of type T_2 in the change and only two variables of type T_1 are available, then ConFix cannot concretize one of the three normalized variables. Hence ConFix decides a type is assignable to another type only if there exist enough variables for assignment.

Finally, the compatibility between normalized and concrete methods is defined by their abstract signatures and assignable types. For each normalized method, ConFix lists up compatible concrete methods and randomly selects one of them to replace the abstract method.

Definition 7 (Method Compatibility) A normalized method m_n and a concrete method m_c are compatible if and only if they have the same abstract signatures and types T_n^i and T_c^j appeared in signatures are all assignable for all $i = j$.

After concrete methods are fixed, ConFix decides concrete types for remaining normalized types, and then selects concrete variables for normalized variables. If abstract types are appeared in abstract signatures of the normalized methods, their concrete types are already fixed. Hence ConFix randomly selects one of the assignable types only for the remaining normalized types. Once all normalized types are replaced with concrete types, the type of each normalized variable is fixed, and ConFix selects one of the concrete variables with the same type randomly.

One remaining part of change concretization is normalized string literals. ConFix collects string literals from buggy code - in the same buggy class - and assigns one of the collected string from the same buggy class to each normalized string. Since there is no type constraints for strings, ConFix only considers the frequency of the collected strings. More specifically, ConFix uses roulette wheel selection [24] with string's frequencies as weights.

Similar to hash match method, ConFix considers different sets of identifiers collected from different scopes. First, ConFix considers identifiers more closely related to the current fix location. This *neighbor* set contains identifiers appeared in neighbor statements and declared method parameters. Neighbor statements include the statement which is currently being modified, and its parent statement as well as statements before and after the statement.

Suppose that ConFix tries to update argument `pos` in the following example from one of the Defects4j bug.

```
for (int pt = 0; pt < consumed; pt++) {
  pos += Character.charCount(Character.codePointAt(input, pos));
}
```

The current statement is an assignment of `pos`, and its parent statement is a for statement. There is no statement before and after it, but if there was any statement right before and after the current statement, ConFix would also collect identifiers from them. To exchange `pos` with another variable, it is reasonable to consider variables appeared in neighbor statements since they are related to functionalities implemented by this area of code. Also, a parameter of a method - `input` in this example - is often used throughout the method, since a method's role is usually taking inputs and processing them with its statements to generate expected outputs. Hence using identifiers from neighbor statements and method parameters can adapt a change more close to the current fix location.

If ConFix cannot find enough concrete names to replace abstract names, it expands the scope to all local variables in the same method and member variables and methods declared in the same class (*local*). Local variables are declared in a method since they are necessary within the method. Similarly, member variables and methods are declared in a class since they are related to the class. Although the relevance may be lower than the first set of identifiers, it is worth to consider using them to adapt a change to the current method or class. Even this attempt fails to find sufficient concrete names, ConFix finally considers any identifiers appeared in the current buggy class (*class*).

3.4 Patch Prioritization Strategy

As ConFix generates patch candidates one by one, it is important to decide which candidate should be generated first. Patch prioritization is important in ConFix because (1) modifying the wrong locations or applying the wrong changes reduces the probability of generating a patch within the time budget; and (2) ConFix terminates execution after finding the first plausible patch [47], instead of generating all possible patch candidates.

We already discussed about the order of covered lines in Section 3.2. In addition, ConFix applies *Tested First* heuristic using failed test class names to sort

suspicious lines. Sometime actual buggy lines have lower scores with SBFL techniques, if they are also executed by passing tests frequently. To mitigate this issue, ConFix predicts classes executed by failed test classes based on their name similarity. The names of failed test classes and classes covered by them are tokenized, and the number of common tokens between each pair of a buggy class and a test class is computed. For instance, for a failed test class `ProcessClosurePrimitivesTest` of package `com.google.javascript.jscomp`, a class `ProcessClosurePrimitives` of the same package is selected since seven tokens - four package names and `Process, Closure, Primitives` - are common, and all the other classes have less common tokens than this class. Hence ConFix lists up all covered lines of the selected class in descending order of the score, then all the other lines are ordered after that in the same descending order.

For each fix location candidate, ConFix applies *maxChangeCount* of changes in the descending order of their frequency. This prioritization simply tries more popular changes in past patches first. The reason for a limited number of change application is to avoid bottleneck and investigates more fix location candidates as explained in Section 3.2.

Since one abstract change may produce different concrete changes based on change concretization, ConFix also tries to concretize the same change *maxTrials* times. For change concretization, ConFix first tries hash match method to generate patch candidates more similar to human-written code fragments. Each collected code fragment is used for concretization in ascending order of its distance to buggy code and descending order of their frequency for the same distance (Section 3.3.1). If there is no available code fragments for hash match method, ConFix applies TC method, which can introduce a new code fragment which have not been appeared in buggy code.

4 Evaluation Design

In this section, we provide information of subjects for evaluation and experimental setup to generate patches with ConFix.

4.1 Subjects

To evaluate ConFix, we need a set of bugs to be fixed, and a set of human-written patches to collect changes for patch generation. Table 1 shows simple statistics of collected human-written patches used for the evaluation.

Table 1: The Information of Subjects for Change Pools

Project	Patches	Changed Files	Project	Patches	Changed Files
collections	79	256	*ivy*	357	691
derby	1,270	3,355	*lucene*	1,446	13,932
groovy	1,535	6,141	*mahout*	265	1,102
hadoop	465	6,479	*pdfbox*	1,004	1,883
hama	64	177	Total	6,485	34,016

We collected changes with PTLRH and PLRT contexts from nine Java open source projects in Apache Software Foundation. We first listed issue numbers categorized as bugs from Apache's issue tracking system[1] for each project. Then we identified bug-fix commits (patches) containing the issue numbers in their commit log. Patches and Changed Files columns show the number of identified human-written patches and changed files in those patches respectively. We collected over 6K human-written patches with about 34K changed files as resources for patch generation.

For a set of bugs, we used 357 real Java bugs in Defects4j 0.1.0 dataset [16,32], a popular benchmark for APR technique evaluation. Defects4j dataset provides bugs and test cases for evaluation of automatic patch generation technique. These bugs were collected from five different software projects, *JFreeChart*, *Closure*, *Commons Lang*, *Commons Math* and *Joda-Time*. It also contains human-written patches for the bugs, hence we can evaluate the quality of generated patches based on these patches.

Note that we selected a completely different set of projects for change resources from the projects in Defects4j dataset. It guarantees that changes from actual human-written patches of Defects4j bugs are not included in ConFix's change pools. Hence ConFix's patch generation was only dependent to general population of changes collected from human-written patches, without influence of the original human patches.

4.2 Experimental Setup

For the evaluation, we applied ConFix to 357 bugs from Defects4j dataset with PTLRH and PLRT change pools, on an Amazon EC2 instance (*m5.xlarge*) with 4 CPUs and 16 GB memory. To prepare the experiments, we first checked for flaky tests by executing all given test cases on both buggy code and human-written patches. Since Defects4j dataset also provides a list of failing tests, these tests should be failed on buggy code and passed on patches. We found that there were 12 Closure bugs whose failing tests were always failed even with human-written patches in our environment. For these bugs, even if ConFix generated an identical patch to a human-written patch, it cannot be verified with the given test cases. Therefore, we did not apply ConFix to these bugs and considered them un-fixed.

After all bugs and their test cases were checked, we ran ConFix for each bug. ConFix first tried to generate at most 20K candidates with PTLRH change pool. If it failed to generate a patch, it tried the same with PLRT change pool. Although ConFix was allowed to generate at most 40K candidates, the actual number of generated candidates might be smaller based on the number of possible fix locations and applicable changes after filtering with contexts.

For each fix location, ConFix tried 25 frequent changes which is the average number of changes per context in PLRT change pool (Table 2). In this way, ConFix can check sufficient changes while skipping some exceptional fix locations with a lot of applicable changes. For each abstract change, ConFix concretized it maximum five times to apply various concrete changes for patches.

[1] Apache's JIRA issue tracker (https://issues.apache.org/jira).

We also set time budget to two hours, which means that ConFix terminated after two hours even if generated candidate number was less than the maximum candidate number. This time budget did not provide significant advantage to ConFix, compared with 1.5 to 5 hours of other compared existing techniques [15,58,60].

To apply fault localization technique, coverage information was necessary. We used the coverage information of 357 bugs and their tests from a previous fault localization study [43] obtained by GZoltar [4] to prevent any advantage or disadvantage about coverage information.

5 Results

5.1 RQ1: Characteristics of Change Pools

From collected human-written patches, we built two change pools with PTLRH and PLRT contexts respectively. Table 2 shows the information of the PTLRH and PLRT change pools.

Table 2: The Information of PTLRH and PLRT Change Pools

	PTLRH	**PLRT**
Total Changes	216,274	216,274
Unique Changes	44,205	44,205
Contexts	38,160	2,474
Context-Change Pairs	76,597	62,310
Avg. Changes per Context	2.01	25.19

In both change pools, ConFix identified about 216K abstract AST subtree changes from the collected patches. Among them, 44,205 changes are unique. AST subtree changes preserve the structure of changed subtrees, hence the number of unique changes is large. If we only consider change type and changed entity type like Change Type and changed Entity Type (CTET) repair model of a previous study [33], there are only 549 different changes, which is much less than the unique changes, but greater than 173 of the previous study. The difference is mainly due to replacements, since there are two changed entity types (i.e., old and new) and more variants. These large number of more specific changes not only provide the information that which kind of changes are popular in bug-fixes, but also supply abstract code fragments which can be directly used for patch generation.

Collected changes are categorized by 38,160 PTLRH contexts and 2,474 PLRT contexts in change pools respectively. If we consider one unique change appeared in different contexts (context change pairs) as different changes, there are 76,597 and 62,310 changes for PTLRH and PLRT change pools respectively. It means that for each PTLRH and PLRT context, there are only 2.01 and 25.19 unique changes which ConFix needs to consider for each context. This is huge reduction from the entire set of 44K unique changes, which greatly saves the effort to select a change to be applied.

Table 3: Change Type Distribution and Frequent Changes

Type	Count	Freq.	CTET	Freq.	Change	Freq.
`insert`	45.03%	44.71%	`update SN`	8.45%	`update var0`	4.81%
`replace`	52.08%	36.55%	`insert MI`	5.81%	`update Type0`	3.68%
`update`	2.89%	18.74%	`insert ASGN`	5.56%	`update method0`	2.82%

`SN`: `SimpleName`, `MI`: `MethodInvocation`, `ASGN`: `Assignment`

Table 3 shows the distribution of change types and frequent changes in the collected changes. Count column shows how many unique changes belong to a certain change type. Freq. columns indicate the ratio of each change to all collected changes. Type, CTET and Change column represent change types, change and entity types, and abstract AST changes respectively.

There are many variants (i.e., different changed subtrees) for insert and replace changes, while update changes occupy less than 3% of the unique changes. Unlike the others, update changes can vary only for values such as update 0 to 1 or 0 to -1, hence unique change number is smaller. However, in terms of frequency, update changes take 18.74% of all collected changes. This indicates that each update is more frequent than the other type changes, which is consistent to the result that the most frequent CTET is `update SimpleName` in Table 3.

In actual abstract subtree changes (Change column), update changes are even more frequent. In CTET, the second and third most frequent changes are insert changes. However, for subtree changes, all top 3 frequent changes are updating variable, type and method. Insertion of method calls or assignments are more diverse when we consider actual inserted subtrees. Hence each change's frequency can be smaller than updates, which represent a single node change with limited variations. This result is consistent to the finding of a previous study that smaller changes are more frequent [41].

Such high frequency of update changes can be an obstacle for change selection based on natural distribution of changes. If we consider each change's frequency as the probability that the change is selected, small changes always have higher priority than bigger changes. If ConFix only considered change's frequency, it would try update changes many times before other various changes are applied to generate patch candidates. In terms of patch generation, this is not desirable.

ConFix leverages local change distribution represented by the frequency of changes in a specific context to resolve this issue. For example, consider a PTLRH context `P:MethodInvocation[arguments],L:{SimpleName},R:`, which represents a method call's argument as a change location. The most frequent change of this context is `insert var0` (Freq. 1,687), which is different from the frequent changes shown in Table 3. The next most frequent change is `update var0` (Freq. 1,056), whose frequency is 37% less than the insertion. In other contexts for statement level changes, update changes are not even appeared in the contexts. This result shows that local change distribution is different from global change distribution, hence ConFix's change selection can be more customized to a specific context.

5.2 RQ2: Bug-Fixing Performance of ConFix and Comparison to Existing Techniques

To evaluate ConFix's performance, we manually inspected all generated patches and assessed their correctness, then compared them with generated patches of three other techniques, ssFix [60], CapGen [58] and SimFix [15].

These three techniques are closely related to ConFix. ssFix adapts collected code fragments to generate patches, which is similar to ConFix using abstract changes preserving the structure of changed code fragments. However, ConFix considers contexts to find changes more probable to generate patches, while ssFix computes syntactic similarities of code fragments to buggy code for the same purpose. CapGen considers three different contexts, and its genealogy context is similar to ConFix using AST contexts. However, both CapGen and SimFix mine high-level abstract changes and concretize them with code fragments obtained from buggy source code. Unlike them, ConFix employs more fine-grained change representation which preserves code structure, hence it can provide new code fragments not in buggy code, and concretizes them with TC method to generate patches which might not be generated by CapGen and SimFix.

During evaluation, we found that correctness assessment of generated patches for the compared techniques was not sufficient. Some of the generated patches only addressed a part of issues compared to human patches, but still they were considered correct. For example, Fig. 6 shows patches by human developers, ConFix and ssFix for C1 bug. Note that bug will be referenced as "Project's Identifier + BugId" (e.g., C1 indicates the first bug of Chart) with project identifiers shown in Table 4.

```
- if (dataset != null) {  //Buggy
+ if (dataset == null) {  //Human, ConFix
-   return result;  //ssFix
  }
```

Fig. 6: Human, ConFix and ssFix patches for C1 bug.

Human and ConFix patches modify the operator `!=` to `==`, while ssFix patch removes the return statement. This patch of ssFix was considered valid, since it does not return `result` when `dataset` is not null and passes all given test cases. However, its behavior is different from the human patch when `dataset` is null, which is actually incorrect.

Hence we evaluated the correctness of generated patches for ConFix as well as the other techniques with the following criteria. We marked a bug is fixed by a generated patch, if the patch is syntactically or semantically equivalent to the human patch. Also, we considered a generated patch is incorrect, if we found a case which the generated patch shows different outcome than the human patch or the generated patch is only partial (e.g., modified only one of the two fix locations in human patch).

Table 4 shows the number of correctly fixed bugs by patches generated by ConFix and other techniques. Note that ConFix only generates one plausible patch for each bug, hence the fixed bug number is identical to the correct patch number. After re-evaluation, if the numbers are different from the originally reported num-

Table 4: Number of Correctly Fixed Bugs

Project	ConFix	ssFix	CapGen	SimFix
Chart(C)	4	2(3)	4	3(4)
Closure(CL)	6	2	N/A	6
Lang(L)	5	5	5	5(9)
Math(M)	6	9(10)	11(12)	12(14)
Time(T)	1	0	0	0(1)
Total	22	18(20)	20(21)	26(34)

bers of ssFix [60], CapGen [58] and SimFix [15], we show the original results in parentheses. For CapGen, Closure was not a subject for its evaluation, hence no patches are available. Individual patch assessment used to compute the numbers are publicly available.[2]

ConFix correctly fixed 22 bugs in total, which is comparable to the existing techniques. In terms of numbers, ConFix generated the most number of correct patches for all projects except Math. For Time, only ConFix generated a correct patch for one bug (T19). SimFix reported that T7 was fixed by a generated patch, but we found that the patch was incorrect. Actually, ConFix generated the exactly same patch for T7 too. For Chart, Closure, and Lang, ConFix generated the same number of correct patches as the other best technique.

For Math, ConFix did not generate many patches compared to the other techniques. We analyzed bugs and their human patches which were fixed by the others but not by ConFix. For these bugs, generated patches are not complicated and usually consist of simple changes, but the changes are not included in change pools with the same context of fix locations. This is a trade-off of using context to reduce the difficulty of change selection. By checking context, ConFix only needs to consider a small number of changes, but sometimes it cannot generate a simple patch since a necessary change is only included in different contexts. However, this issue can be addressed by collecting more changes, if ConFix can find necessary changes effectively.

The number of fixed bugs does not represent the fixed bug set. For instance, all four techniques fixed five bugs in Lang, but it is not necessary that they fixed the same five bugs. If ConFix fixes bugs which have not been fixed by the other techniques, ConFix can be complimentary and worth to use with others.

Table 5: Bugs Fixed by ConFix Only and Also Fixed by Compared Techniques

Project	ConFix Only	Also Fixed By Others
Chart(C)	C10	C1, C11, C24
Closure(CL)	CL38, CL92, CL93, CL109	CL14, CL73
Lang(L)	L24, L51	L6, L26, L57
Math(M)	M34	M5, M30, M33, M70, M75
Time(T)	T19	None
Total	9 Bugs	13 Bugs

Table 5 shows the full list of bugs fixed by ConFix. “ConFix Only” column shows bugs fixed by ConFix only. “Also Fixed By Others” Column represents bugs fixed by ConFix, as well as at least one of the other techniques.

[2] https://github.com/thwak/confix2019result/

There are nine bugs which are only fixed by ConFix. ConFix fixed at least one bug for all projects which was not fixed by the others. We also checked bugs reported as fixed by other existing techniques (jGenProg [23], jKali [47], Nopol [9], ACS [62], HDRepair [20], Jaid [5], AVATAR [25]). We inspected fixed bugs in [32], [60], [62], [5] and [25], to obtain the result for all 357 bugs, since each study only contains results for a part of the techniques or the bugs. For AVATAR, we used results for Supplemented_FL-based APR [25], since it assumed the same configuration for fault localization as ConFix. If any of them reported a bug was fixed by a technique, we considered the bug is fixed by the technique. Based on the inspection, we found that L24 (Jaid, ACS), L51 (Jaid, HDRepair) and CL38 (AVATAR) are also fixed by the others. Still, ConFix fixed six bugs which have not been fixed by the existing techniques. This result indicates that ConFix is capable of generating patches which cannot be found by the existing techniques, and it can complement the others.

5.3 RQ3: The Effectiveness of Search Space Navigation

ConFix uses various strategies to effectively navigate its patch search space. It filters out candidate fix locations and changes to narrow down search areas, and leverages frequencies of contexts and changes to decide which area should be searched first. To concretize abstract changes, it also collects code fragments and identifiers from different range of code. We analyzed how these strategies contribute to fix the bugs.

5.3.1 Precision and Recall

First, we computed precision and recall of ConFix. Precision is the ratio of correct patches to all generated patches. Recall represents that how many patches included in ConFix's search space are actually found.

ConFix generated 92 plausible patches and 22 of them are correct. The precision is 23.9%, which is lower than ssFix (30.0%, 18/60), SimFix (46.4%, 26/56) and CapGen (80.0%, 20/25). Since ConFix leverages over 62K to 76K changes, the probability that one of these changes accidentally generates a plausible, but incorrect patch is quite high. This overfitting issue [53] is a weakness of ConFix since its ability to generate various patch candidates comes from a large set of collected changes. However, there have been several studies [55,59,61,63] to mitigate the issue, and ConFix can benefit from such studies.

To compute recall, we identified which bugs have patches in ConFix's search space. We first extracted changes from human patches in the same way we collected changes for changes pools. If a bug's human patch consists of one single change, and the change is included in PTLRH or PLRT change pool, this patch is included in ConFix's search space. Of course, it is possible that a bug can be fixed by a change different from an extracted change. However, identifying all possible fixes is infeasible, hence we only consider explicit changes identical to human patches, which is a reasonable approximation.

Table 6 shows the number of bugs whose patches in the search space (Total) and how many of them have been correctly fixed by ConFix (Correct). Incor-

Table 6: Number of Patches in ConFix's Search Space and Fixed Bugs

Project	Total	Correct	Inorrect	No Patch
Chart(C)	4	4	0	0
Closure(CL)	7	3	0	4
Lang(L)	7	5	1	1
Math(M)	4	4	0	0
Time(T)	1	1	0	0
Total	23	17	1	5

rect column indicates that ConFix generated incorrect patches for bugs, and "No Patch" column represents that ConFix failed to generate a plausible patch.

There are 23 bugs whose patches are in ConFix's search space, and ConFix fixed 17 of them, which leads to 73.9% recall. It means that ConFix can correctly fix about three-quarters of bugs, if necessary changes are included in change pools. Therefore, we can expect more correct patches by providing more collected changes due to ConFix's effective search space navigation. Note that there is only one bug which has a patch in the search space, but ConFix generated an incorrect patch. This result indicates that ConFix's patch prioritization effectively ranks existing correct patches to make them found as the first plausible patch for each bug.

For the five bugs without patches, we analyzed which features of ConFix is not effective for successful patch generation. We found that the failure is closely related to fix location identification and prioritization for 4 out of 5 bugs. Among these four bugs, ConFix failed to fix three Closure bugs (CL1, CL10, CL66), since the right fix location has very low suspiciousness score. ConFix consumed all time budget and assigned patch candidates before it even tried to modify the right fix location. The remaining bug L29 has a patch updating the return type of a method, but this code line is not included in the coverage of test cases. We used coverage information from [43], which was obtained by GZoltar [4]. However, a code line corresponds to a method declaration was not treated as executed, hence the fix location was not even on the list of candidate locations.

These fix location issues are more close to the limitation of SBFL technique rather than ConFix's patch generation strategy. This issue can be resolved by increasing candidate limitation, or providing missing coverage information. We applied ConFix with increased time budget and the maximum candidate number, and confirmed that ConFix generated correct patches for the Closure bugs. L29 bug was also correctly fixed after we added the modified line in human patch into the coverage information.

The one remaining bug with no patch has high patch generation difficulty. It requires a replacement of a string literal with another string literal. The required string literal was not appeared in codebase, hence ConFix could not generate a correct patch. However, any existing techniques leveraging specific code fragments from buggy code may suffer the same issue, since generating a specific string literal can be a tricky problem.

5.3.2 Fix Locations

In addition to the SBFL technique's suspiciousness score, ConFix uses several strategies to further reduce and prioritize fix location candidates. We analyzed

whether these strategies were actually useful to correctly fix the bugs. As a baseline, we consider a strategy (*FL*) which considers all fix location candidates with suspiciousness score greater than 0. In FL strategy, lines are sorted in descending order of scores, and the same score lines are sorted in ascending order of line numbers. Fix location candidates on the same line are sorted in depth-first order.

For analysis, we come up with three strategies *CTX*, *FLFreq*, and *TestedFirst*. CTX strategy further filters out fix locations with no applicable change in change pools. FLFreq strategy prioritizes lines and fix locations as explained in Section 3.2, without special treatment for suspicious classes. TestedFirst is the actual strategy with prioritizing suspicious classes, which was used by ConFix to generate patches (Section 3.4). We compared these four strategies for the number of checked lines and fix locations until each strategy reaches to the actual fixed line or location of a patch.

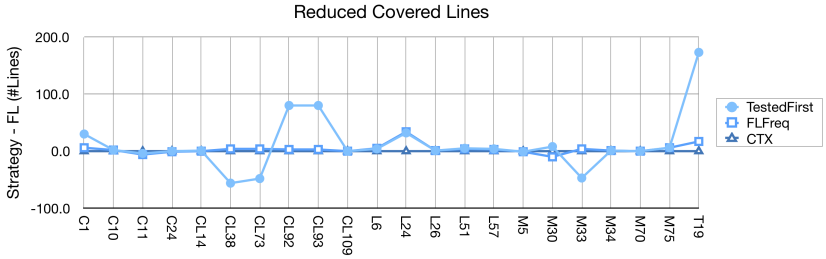


Fig. 7: Reduced Number of Checked Lines for Fixed Bugs

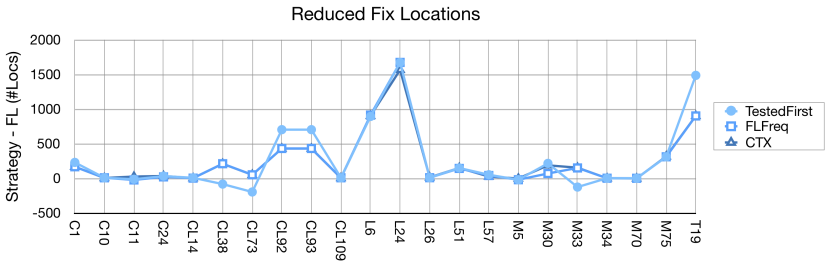


Fig. 8: Reduced Number of Checked Fix Locations for Fixed Bugs

Fig. 7 and 8 represent reduced number of checked lines and locations for bugs respectively. A positive number indicates that each strategy tried less number of lines or locations than FL strategy until it actually found the fix location. In Fig. 7, CTX shows the straight line at 0, which means that it always checked the same number of lines as FL. CTX only filters out lines and locations from FL if there is no applicable change, but the order of lines and locations are the same. This result indicates that there is at least one fix location candidate for every lines with applicable changes, and using CTX alone does not helpful to reduce the number of checked lines.

However, although CTX examined the same number of lines, the number of checked locations is significantly reduced. CTX checked less locations than FL except for one bug. For the one exception, FL and CTX both checked just one

location to generate a patch. The average reduction of checked locations is 48%, which means that ConFix only needs to check about a half of all fix location candidates on average. Therefore, using context effectively reduced search area and contributed to ConFix's patch generation.

FLFreq strategy shows improved performance in both checked lines and locations. In terms of lines, FLFreq checked 1-34 less lines for 15 bugs, but 1-10 more lines for four bugs than FL and CTX. FLFreq basically works as a tie-breaker, which only re-orders lines if they have the same suspiciousness score computed by Ochiai, hence the improvement is limited for lines. In terms of locations, the performance improvement is a bit mixed. To assess the effectiveness of frequency-based prioritization, we compared FLFreq with CTX. FLFreq checked less locations than CTX for 12 out of 22 bugs, but checked more locations for the remaining 10 bugs. The differences between FLFreq and CTX strategies - at most 118 locations - are smaller than CTX and FL, since CTX checked 256 less locations than FL on average. These results show that the reduction of checked lines and fix locations are mainly due to contexts, but using frequency is still effective for some cases. Note that we did not include a graph for the case if ConFix did not sort locations on each line in ascending order of change count 3.2. We found that if this strategy was not used, ConFix checked average 0.41 more locations, which is negligible.

TestedFirst shows mixed performance in both lines and locations. Since TestedFirst speculates classes under test and tries to modify them first, the improvement highly depends on the correctness of speculation. In Fig. 7 and 8, TestedFirst lines are clearly above FLFreq for C1, CL92, CL93, M30, T19 bugs, which means that the speculation was useful. There exist other bugs CL38, CL73, M33 that TestedFirst picked incorrect classes and needed to check more lines and locations than FLFreq. In total, TestedFirst checked less locations than FLFreq for 10 bugs while it checked more locations for only five bugs. Hence applying TestedFirst was not harmful for 77% of the fixed bugs. If we compare TestedFirst to the baseline (FL), it checked average 65% less locations for 17 bugs, which proves that ConFix's strategy is quite effective compared to a strategy only relying on suspiciousness scores.

In addition, we found that L24 and T19 bugs would not be fixed without the reduction of fix locations due to TestedFirst strategy. Note that we limited the number of patch candidates to 20K for each change pool. Since ConFix applies at most 25 changes to each location at most five times, it is possible that ConFix exceeds the limit before it tries to modify the actual fix location. For L24 bug, ConFix (i.e., TestedFirst) generated 24,231 patch candidates (20K for PTLRH, 4,231 for PLRT) to fix the bug, while FLFreq only reached to the same fix location after generating 44,431 (20K for PTLRH, 24,431 for PLRT). For T19 bug, the 9th bug which ConFix generated was the correct patch, while FLFreq required 20,900 patches to reach the fix location. With our experiment settings, these two bugs could not be fixed if ConFix used FLFreq strategy.

5.3.3 Change Selection

For each fix location candidate, ConFix applies changes with the same context in descending order of their frequency. We analyzed changes used to generate patches to assess this change selection strategy was effective. First, we identified changes used to fix bugs and their change pools. We found that 19 out of 22 (86%) bugs

were fixed by changes from PTLRH change pool, and the rest three bugs were fixed by PLRT change pool. Note that PTLRH and PLRT changes pools have the same set of unique changes. Theoretically, patches generated with PTLRH change pool can also be generated by PLRT change pool, but it will require much more resources. If PTLRH contexts do not overly filter out necessary changes, it is more efficient than PLRT, since ConFix needs to check only two changes on average for each location, which is much lower than 25 changes of PLRT. Therefore, this result indicates that using PTLRH context is effective and efficient, and some missing patches were still found by ConFix's extended search with PLRT context.

ConFix also leverages local change distribution to prioritize changes. To assess the effectiveness of local distribution, we compared changes' local ranks (rank in a context) with global ranks (rank in all collected changes) based on frequency. ConFix used 15 different changes in 21 different contexts for 22 patches. This result shows that ConFix applied various changes for patch generation, and did not favor a few popular changes. ConFix ranked the changes at top 1-15 in a list of changes retrieved for the fix location of the patches. For 14 out of 22 (63.6%) patches, ConFix ranked the applied changes at the top, and it ranked the changes within top-3 for 81.8% of the patches.

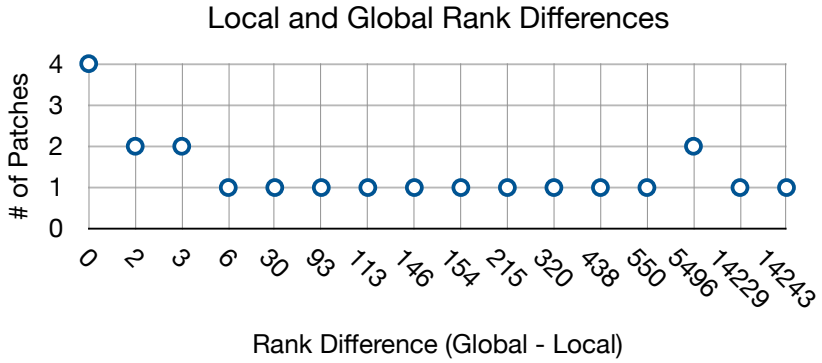


Fig. 9: Differences in Local and Global Change Ranks

Fig. 9 shows the differences of local and global ranks of the changes. A change's global rank indicates its frequency rank in all unique collected changes. If a change has global rank 10, it means that this change is the 10th most frequent change among all collected changes. There are only four patches whose change's local rank is identical to global rank. All same rank patches used `update var0 to var0` in different contexts, which is the most frequent change globally (Table 3). For 13 out of 22 (59.1%) patches, the rank difference is at least 30, which is even higher than the maximum number of changes ConFix applied for each location. These low global rank changes might not be found if they were chosen from all changes based on their frequency.

One interesting observation is that sorting local changes with their global frequency does not show significant difference. Originally, ConFix sorted changes in a certain context, based on their frequency on that context only. We re-ordered the changes based on their global frequency, but ranks of the changes used to fix bugs were not altered significantly. Using local frequency placed changes for 4 out of 22

patches at 1-3 higher position than global frequency, while it ranked 1-3 position lower for changes of 5 out of 22 patches. For the other 13 patches, change ranks were not changed, since most of them are already at rank 1, except for one change which has rank 16. The situation is similar for all generated patches, since local frequency provides higher ranks to necessary changes for 23 patches, while it gives lower ranks for 22 patches and the same ranks for the rest.

This result indicates that the performance improvement is mainly due to filtering of changes based on context. Once changes have been filtered out, about 82% of the changes necessary to fix bugs are ranked within top-3 whether local or global frequency is used. It also means that ordering changes based on their frequency itself (either local or global) is also an effective strategy for change selection, since the changes are at high rank although there are about 18 changes on average for each fix location of the correct patches.

5.3.4 Change Concretization

Table 7: Number of Patches for Concretization Methods

Methods	*hash-class*	*hash-package*	*neighbor*	*local*	*class*	*none*	Total
Correct	8	1	4	3	1	5	22
Incorrect	22	2	23	4	5	14	70
Total	30	3	27	7	6	19	92

We inspected how changes were concretized for all plausible patches. In Table 7, Methods row shows concretization method and the scope of material collection which was explained in Section 3.3. Hash-class and hash-package indicate concretization with hash matched code fragments from a buggy class and package respectively. Type compatible concretization is divided into three categories - neighbor, local, and class, which represent scope where concrete identifiers were collected. *none* indicates a change without normalized values, which does not require concretization, such as updating 0 to 1.

Overall, hash match (33, 35.9%) and TC methods (40, 43.5%) take about 80% of all generated patches. The remaining 20% of the patches were generated by changes without concretization. Note that ConFix only expands search area for concretization if there is no material in the current scope. Hence methods with narrower scope might contribute to more patches. However, every methods are responsible for at least one correct patch, which means that ConFix's search area extending strategy works effectively.

Interestingly, hash match methods take higher portion than TC methods, while there are more patches generated with TC methods for incorrect or all patches. Hash match methods generated 9 out of 22 (40.9%) correct patches, and TC methods produced 8 out of 22 (36.4%) patches. This is reversed in incorrect patches since hash match takes 34.3% and TC is responsible for 45.7%. Based on this result, we can expect more correct patches when hash match methods are used.

However, it does not mean that we should not use TC methods, since they generated almost same number of correct patches as hash match. Hash match methods can generate patches only if there exists a code fragment in buggy code which can be a part of the patch, similar to CapGen and SimFix which use high-

level changes and concretize them with code fragments in buggy code. On the other hand, TC methods collect concrete names from buggy code and generate a patch with a code fragment which has not been appeared in the buggy code. Because ConFix's concretization strategy applies hash match methods first and tries TC methods later, this strategy can exploit the advantages of both methods.

6 Threats To Validity

Errors which might reside in our implementation of ConFix and nondeterministic features of the technique could be one validity threat to this study. For instance, TC methods uses random and roulette wheel selections, hence repeated experiments may provide different results from the results reported in the study. However, the majority of the correct patches were generated without TC methods (Table 7), and these patches can be generated again in repetitions, since the same prioritization will be applied for patch generation. Also, we carefully implemented ConFix that a random seed value controls all nondeterministic features of ConFix, hence replication of our experiment is possible when the same seed is given. Our implementation used for evaluation is also publicly available. [3]

Another concern is that our evaluation results might be different if we used other collected changes from different human-written patches. However, to prevent a human-written patch for a bug is included in the collected changes, we did not choose any projects in Defects4j dataset for change collection. Also, ConFix has high recall (Section 5.3.1), which means that we might also obtain even improved results if we collected more patches for change pools. Hence selection of patches does not undermine our evaluation results significantly.

Manual assessment of patches could be another issue, since we do not have domain knowledge and the judgement about patch might be subjective and biased. We tried to be conservative and carefully compared generated patches with corresponding human-written patches, but we cannot perfectly guarantee that there is no mistake. However, as we explained in Section 5.2, we set up certain criteria to decide whether a patch is correct or incorrect. All ConFix-generated patches, their correctness assessment and other related information are publicly available. [4] The correctness assessment also includes the detailed reasons of our judgement on ConFix-generated correct patches, and why we decided some of the patches generated by the other techniques were actually incorrect.

7 Related Work

7.1 Automatic Program Repair

Many APR techniques leverage human-written code and bug-fixes collected from software repositories. We already discussed about ssFix [60], CapGen [58] and SimFix [15] in Section 5.2.

There are other APR techniques which leverage human patches or mined fix patterns. Genesis [27] infers code transforms for certain defect classes such as NPE

[3] `https://github.com/thwak/ConFix`

[4] `https://github.com/thwak/confix2019result`

or class cast issue from human-written patches and uses them to generate patches. The search space of Genesis is inferred by integer linear programming, which maximizes the number of validation patches generated by a set of code transforms. Then it applies all applicable code transforms to each fix location candidate to generate a patch. In addition to changes, ConFix explicitly defines and uses AST contexts to find fix locations and select necessary changes more effectively. ELIXIR [51] uses method calls with local variables, fields, or constants to synthesize patches. It also leverages the machine-learnt model using code surrounding buggy locations and bug reports. The approach is similar to ConFix in high level idea which exploits code surrounding fix locations and local variables or constants. ConFix's TestedFirst prioritization may play similar role as using information from bug reports. However, ConFix does not favor certain popular changes (Section 5.3.3), but it applies various changes if they belong to the same context. Prophet [30] and HDRepair [20] leverage human patches to prioritize modifications for patch candidates with pre-defined mutations. Although their prioritization methods are effective, their ability to generate various patch candidates is limited compared to ConFix, which can employ a huge number of changes collected from human-written patches. PAR [18], SPR [28], and relifix [54] also use pre-defined fix templates obtained from human intuition or manual inspection on human-written patches, and have the same limitations unlike the techniques which can provide more changes or code fragments from a collection of software repositories. AVATAR [25] uses fix patterns of static analysis violation to obtain more reliable set of bug fixes. One of the key ideas of AVATAR is polishing a set of patches by executing static bug detection tool in pre-processing. Such polishing could be useful to increase ConFix's precision by removing unprofitable changes.

There exist a line of techniques which are known as semantics-based program repair techniques [7, 17, 21, 35, 36, 42]. These techniques use semantic analysis and a given test suite to obtain semantics of a buggy program and synthesize code to satisfy some constraints identified from the semantics. S3 [21] tried to address the patch overfitting [22, 53] and scalability issues of semantic-based repair by constrained search space and ranking functions. ConFix also tries to constrain its search space using AST contexts, hence its context-based change application can collaborate with S3's ideas to synthesize patches more efficiently. SearchRepair [17] is an automatic program repair technique which generates patches with semantic code search. ConFix and SearchRepair both try to leverage human-written patches for new patch generation, but the difference is that SearchRepair uses SMT constraints to find alternative code fragments to fix bugs, while ConFix uses syntactic context to guide a search for appropriate changes. Identifying AST contexts is less costly than deriving SMT constraints, but these contexts still provide syntactic information which also imply some of program semantics, although it does not fully represent program's semantics like SMT constraints.

There are more techniques trying a semantic-based repair [9, 35, 36, 42], which usually leverages SMT constraints solvers to generate bug fixes. Although such techniques use a quite different method, ConFix and these techniques can be complementary to each other. These semantic-based repair techniques often synthesize new expressions for branch conditions or assignments. For instance, ConFix may use alternative concretization strategy which refers to semantic-based repair's synthesized expressions to concretize changes. Semantic-based repairs also may obtain

some hints for the structure of new expressions needed to be synthesized for patch generation.

Many other automatic program repair techniques using generated-and-validate approach also have been proposed [1,8,9,13,23,44–47,56,57]. ConFix differs from these previous techniques due to the point that it can automatically collect abstract individual changes in large scale and it uses them to generate patch candidates, instead of generating patch candidates with several pre-defined modifications or mutation operations with limited resource of code fragments.

7.2 Studies on Human-written Patches

There are studies to reveal various characteristics of human-written patches and their usefulness for patch generation. Nguyen et al. found that changes in bug fixes are repetitive, and smaller changes are even more repetitive [41]. The rational behind ConFix is consistent to this finding, which implies that changes collected from human-written patches can be repeatedly used for patches of other bugs. We also design the change application technique to collect small individual changes from one patch to increase repetitiveness. Martinez et al. proposed a repair model based on repair actions collected from human-written patches [33]. They tested their repair model in their simulations to recommend changes necessary for patch generation. We further develop this idea and introduce ConFix, which is a complete technique to generate source code level patches with changes collected from human-written patches. Zhong et al. studied real bug fixes, and found that bug fixes often require multiple dependent changes [65]. ConFix does not address this issue yet, but it shows impressive results with single change patches, and it could be a starting point to be extended to generating patches with multiple changes. There is another study investigating the usefulness of past fixes in composing new fixes [64]. This study revealed that an APR approach which composes new fixes by reusing code structures of past fixes can show promising performance. ConFix also collects abstract changes preserving the structure of changed code and applies them to generate new patches.

There exist other studies on changes and source code's uniqueness which imply the potential of techniques leveraging existing code fragments or changes [2,12,34,49]. Empirical evaluation of ConFix and fixability analysis results provide similar implication that we can obtain necessary changes for new bug-fixes from existing patches.

7.3 Change Collection and Application

There are studies about code transfer or mining bug-fix patterns which might play a similar role as the context-based change application technique [3,26,31,37–39,50]. The key role of the change application technique in ConFix is collecting changes from human-written patches for new patch candidate generation. Hence we may consider to use these code transfer techniques to develop new methods for patch candidate generation in ConFix.

Another line of work we may discuss is AST differencing techniques. Although we employs a specific technique, other AST differencing techniques [10,11,48] can

be also used to obtain individual changes from source code patches written by human developers. In this case, it may be required to adjust our design of change collection and application for a new differencing technique. However, it is possible to apply the high level ideas such as collecting abstract changes with their AST contexts regardless of adjustment. Since the context is defined by AST nodes near a changed AST subtree, the technique can identify similar contexts from a changed AST subtree identified by other AST differencing techniques.

8 Conclusion

In this paper, we introduce ConFix, an automatic patch generation technique leveraging human-written patches with our context-based change application technique used by ConFix. In the evaluation on 357 bugs from Defects4j dataset, ConFix correctly fixed 22 bugs with generated patches. We also found that 6 out of 22 bugs are not fixed by the compared existing techniques. Our analysis on ConFix's patch generation strategy shows that ConFix effectively explores its search space to generate correct patches.

Although we demonstrated the promising results of automatic patch generation using changes collected from human-written patches, still there exist some opportunities for improvement. For example, we may try to generate patches with multiple changes to improve partial patches up to acceptable patches, or use more sophisticated concretization methods to effectively generate high quality patches. We hope these opportunities will be explored in future work.

References

1. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on, pp. 162–168 (2008). DOI 10.1109/CEC.2008.4630793
2. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The Plastic Surgery Hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '14 (2014)
3. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 257–269. ACM, New York, NY, USA (2015). DOI 10.1145/2771783.2771796. URL http://doi.acm.org/10.1145/2771783.2771796
4. Campos, J., Riboira, A., Perez, A., Abreu, R.: Gzoltar: an eclipse plug-in for testing and debugging. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 378–381 (2012). DOI 10.1145/2351676.2351752
5. Chen, L., Pei, Y., Furia, C.A.: Contract-based program repair without the contracts. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 637–647 (2017). DOI 10.1109/ASE.2017.8115674
6. Chilowicz, M., Duris, E., Roussel, G., Paris-est, U.: Syntax tree fingerprinting: a foundation for source code similarity detection (2009)
7. D'Antoni, L., Samanta, R., Singh, R.: Qlose: Program repair with quantitative objectives. In: S. Chaudhuri, A. Farzan (eds.) Computer Aided Verification, pp. 383–401. Springer International Publishing, Cham (2016)
8. Debroy, V., Wong, W.E.: Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10, pp. 65–74. IEEE Computer Society, Washington, DC, USA (2010). DOI 10.1109/ICST.2010.66. URL http://dx.doi.org/10.1109/ICST.2010.66

9. DeMarco, F., Xuan, J., Le Berre, D., Monperrus, M.: Automatic repair of buggy if conditions and missing preconditions with smt. In: Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, pp. 30–39. ACM (2014)
10. Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Montperrus, M.: Fine-grained and Accurate Source Code Differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 313–324. ACM, New York, NY, USA (2014). DOI 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982
11. Fluri, B., Wursch, M., Pinzger, M., Gall, H.: Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. Software Engineering, IEEE Transactions on **33**(11), 725–743 (2007). DOI 10.1109/TSE.2007.70731
12. Gabel, M., Su, Z.: A Study of the Uniqueness of Source Code. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp. 147–156. ACM, New York, NY, USA (2010). DOI 10.1145/1882291.1882315. URL http://doi.acm.org/10.1145/1882291.1882315
13. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012). DOI 10.1109/TSE.2011.104
14. Hill, A., Păsăreanu, C.S., Stolee, K.T.: Automated program repair with canonical constraints. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, pp. 339–341. ACM, New York, NY, USA (2018). DOI 10.1145/3183440.3194999. URL http://doi.acm.org/10.1145/3183440.3194999
15. Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pp. 298–309. ACM, New York, NY, USA (2018). DOI 10.1145/3213846.3213871. URL http://doi.acm.org/10.1145/3213846.3213871
16. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 437–440. ACM, New York, NY, USA (2014). DOI 10.1145/2610384.2628055. URL http://doi.acm.org/10.1145/2610384.2628055
17. Ke, Y., Stolee, K.T., Goues, C.L., Brun, Y.: Repairing programs with semantic code search (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 295–306 (2015). DOI 10.1109/ASE.2015.60
18. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE'13 (2013). URL http://dl.acm.org/citation.cfm?id=2486788.2486893
19. Kim, J., Kim, S.: Location Aware Source Code Differencing for Mining Changes. Tech. rep., Hong Kong University of Science and Technology (2016). URL https://github.com/thwak/LAS. [Online; accessed 05-Mar-2019]
20. Le, X.B., Lo, D., Goues, C.L.: History driven program repair. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 01, pp. 213–224 (2016). DOI 10.1109/SANER.2016.76. URL doi.ieeecomputersociety.org/10.1109/SANER.2016.76
21. Le, X.B.D., Chu, D.H., Lo, D., Le Goues, C., Visser, W.: S3: Syntax- and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 593–604. ACM, New York, NY, USA (2017). DOI 10.1145/3106237.3106309. URL http://doi.acm.org/10.1145/3106237.3106309
22. Le, X.B.D., Thung, F., Lo, D., Goues, C.L.: Overfitting in semantics-based automated program repair. Empirical Software Engineering **23**(5), 3007–3033 (2018). DOI 10.1007/s10664-017-9577-2. URL https://doi.org/10.1007/s10664-017-9577-2
23. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 3–13. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337225
24. Lipowski, A., Lipowska, D.: Roulette-wheel selection via stochastic acceptance. Physica A: Statistical Mechanics and its Applications **391**(6), 2193 – 2196 (2012). DOI https://doi.org/10.1016/j.physa.2011.12.004. URL http://www.sciencedirect.com/science/article/pii/S0378437111009010

25. Liu, K., Koyuncu, A., Kim, D., F. Bissyandé, T.: AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering, pp. 456–467. IEEE (2019)
26. Livshits, B., Zimmermann, T.: Dynamine: Finding common error patterns by mining software revision histories. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pp. 296–305. ACM, New York, NY, USA (2005). DOI 10.1145/1081706.1081754. URL http://doi.acm.org/10.1145/1081706.1081754
27. Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 727–739. ACM, New York, NY, USA (2017). DOI 10.1145/3106237.3106253. URL http://doi.acm.org/10.1145/3106237.3106253
28. Long, F., Rinard, M.: Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 166–178. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786811. URL http://doi.acm.org/10.1145/2786805.2786811
29. Long, F., Rinard, M.: An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 702–713. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884872. URL http://doi.acm.org/10.1145/2884781.2884872
30. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16, pp. 298–312. ACM, New York, NY, USA (2016). DOI 10.1145/2837614.2837617. URL http://doi.acm.org/10.1145/2837614.2837617
31. Martinez, M., Duchien, L., Monperrus, M.: Automatically extracting instances of code change patterns with ast analysis. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, pp. 388–391. IEEE Computer Society, Washington, DC, USA (2013). DOI 10.1109/ICSM.2013.54. URL http://dx.doi.org/10.1109/ICSM.2013.54
32. Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M.: Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. Springer Empirical Software Engineering (2016). DOI 10.1007/s10664-016-9470-4. URL https://hal.archives-ouvertes.fr/hal-01387556/document
33. Martinez, M., Monperrus, M.: Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering **20**(1), 176–205 (2015). DOI 10.1007/s10664-013-9282-8. URL http://dx.doi.org/10.1007/s10664-013-9282-8
34. Martinez, M., Weimer, W., Monperrus, M.: Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 492–495. ACM (2014)
35. Mechtaev, S., Yi, J., Roychoudhury, A.: Directfix: Looking for simple program repairs. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 448–458 (2015). DOI 10.1109/ICSE.2015.63
36. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 691–701. ACM, New York, NY, USA (2016). DOI 10.1145/2884781.2884807. URL http://doi.acm.org/10.1145/2884781.2884807
37. Meng, N., Kim, M., McKinley, K.S.: Sydit: Creating and Applying a Program Transformation from an Example. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pp. 440–443. ACM, New York, NY, USA (2011). DOI 10.1145/2025113.2025185. URL http://doi.acm.org/10.1145/2025113.2025185
38. Meng, N., Kim, M., McKinley, K.S.: Systematic editing: Generating program transformations from an example. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pp. 329–342. ACM, New York, NY, USA (2011). DOI 10.1145/1993498.1993537. URL http://doi.acm.org/10.1145/1993498.1993537
39. Meng, N., Kim, M., McKinley, K.S.: Lase: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 502–511. IEEE Press (2013)

40. Meyer, A.d.S., Garcia, A.A.F., Souza, A.P.d., Souza Jr, C.L.d.: Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). Genetics and Molecular Biology **27**(1), 83–91 (2004)
41. Nguyen, H.A., Nguyen, A.T., Nguyen, T.T., Nguyen, T., Rajan, H.: A study of repetitiveness of code changes in software evolution. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 180–190 (2013). DOI 10.1109/ASE.2013.6693078
42. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 772–781. IEEE Press (2013)
43. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 609–620. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/ICSE.2017.62. URL https://doi.org/10.1109/ICSE.2017.62
44. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al.: Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 87–102. ACM (2009)
45. Qi, Y., Mao, X., Lei, Y.: Efficient automated program repair through fault-recorded testing prioritization. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13, pp. 180–189. IEEE Computer Society, Washington, DC, USA (2013). DOI 10.1109/ICSM.2013.29. URL http://dx.doi.org/10.1109/ICSM.2013.29
46. Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C.: The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering, pp. 254–265. ACM (2014)
47. Qi, Z., Long, F., Achour, S., Rinard, M.: An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pp. 24–36. ACM, New York, NY, USA (2015). DOI 10.1145/2771783.2771791. URL http://doi.acm.org/10.1145/2771783.2771791
48. Raghavan, S., Rohana, R., Leon, D., Podgurski, A., Augustine, V.: Dex: a semantic-graph differencing tool for studying changes in large code bases. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., pp. 188–197 (2004). DOI 10.1109/ICSM.2004.1357803
49. Ray, B., Nagappan, M., Bird, C., Nagappan, N., Zimmermann, T.: The Uniqueness of Changes: Characteristics and Applications. Tech. rep., Microsoft Research Technical Report (2014)
50. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 404–415. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/ICSE.2017.44. URL https://doi.org/10.1109/ICSE.2017.44
51. Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: Elixir: Effective object oriented program repair. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 648–659. IEEE Press, Piscataway, NJ, USA (2017). URL http://dl.acm.org/citation.cfm?id=3155562.3155643
52. Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M.: Automatic error elimination by horizontal code transfer across multiple applications. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15, pp. 43–54. ACM, New York, NY, USA (2015). DOI 10.1145/2737924.2737988. URL http://doi.acm.org/10.1145/2737924.2737988
53. Smith, E.K., Barr, E.T., Le Goues, C., Brun, Y.: Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 532–543. ACM, New York, NY, USA (2015). DOI 10.1145/2786805.2786825. URL http://doi.acm.org/10.1145/2786805.2786825
54. Tan, S.H., Roychoudhury, A.: Relifix: Automated repair of software regressions. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pp. 471–482. IEEE Press, Piscataway, NJ, USA (2015). URL http://dl.acm.org/citation.cfm?id=2818754.2818813

55. Tan, S.H., Yoshida, H., Prasad, M.R., Roychoudhury, A.: Anti-patterns in search-based program repair. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 727–738. ACM, New York, NY, USA (2016). DOI 10.1145/2950290.2950295. URL http://doi.acm.org/10.1145/2950290.2950295
56. Weimer, W., Fry, Z.P., Forrest, S.: Leveraging program equivalence for adaptive program repair: Models and first results. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pp. 356–366. IEEE (2013)
57. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, pp. 364–374 (2009)
58. Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.C.: Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 1–11. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180233. URL http://doi.acm.org/10.1145/3180155.3180233
59. Xin, Q., Reiss, S.P.: Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pp. 226–236. ACM, New York, NY, USA (2017). DOI 10.1145/3092703.3092718. URL http://doi.acm.org/10.1145/3092703.3092718
60. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 660–670. IEEE Press, Piscataway, NJ, USA (2017). URL http://dl.acm.org/citation.cfm?id=3155562.3155644
61. Xiong, Y., Liu, X., Zeng, M., Zhang, L., Huang, G.: Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 789–799. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3180182. URL http://doi.acm.org/10.1145/3180155.3180182
62. Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering, ICSE '17, pp. 416–426. IEEE Press, Piscataway, NJ, USA (2017). DOI 10.1109/ICSE.2017.45. URL https://doi.org/10.1109/ICSE.2017.45
63. Yang, J., Zhikhartsev, A., Liu, Y., Tan, L.: Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 831–841. ACM, New York, NY, USA (2017). DOI 10.1145/3106237.3106274. URL http://doi.acm.org/10.1145/3106237.3106274
64. Zhong, H., Meng, N.: Towards reusing hints from past fixes: An exploratory study on thousands of real samples. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, pp. 885–885. ACM, New York, NY, USA (2018). DOI 10.1145/3180155.3182550. URL http://doi.acm.org/10.1145/3180155.3182550
65. Zhong, H., Su, Z.: An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pp. 913–923. IEEE Press, Piscataway, NJ, USA (2015). URL http://dl.acm.org/citation.cfm?id=2818754.2818864