




# The effectiveness of context-based change application on automatic program repair

Jindae Kim<sup>1</sup>  · Jeongho Kim<sup>2</sup> · Eunseok Lee<sup>2</sup> · Sunghun Kim<sup>1</sup>

Published online: 2 September 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

## Abstract

An Automatic Program Repair (APR) technique is an implementation of a repair model to fix a given bug by modifying program behavior. Recently, repair models which collect source code and code changes from software history and use such collected resources for patch generation became more popular. Collected resources are used to expand the patch search space and to increase the probability that correct patches for bugs are included in the space. However, it is also revealed that navigation on such expanded patch search space is difficult due to the sparseness of correct patches in the space. In this study, we evaluate the effectiveness of Context-based Change Application (CCA) technique on change selection, fix location selection and change concretization, which are the key aspects of navigating patch search space. CCA collects abstract subtree changes and their AST contexts, and applies them to fix locations only if their contexts are matched. CCA repair model can address both search space expansion and navigation issues, by expanding search space with collected changes while narrowing down search areas in the search space based on contexts. Since CCA applies changes to a fix location only if their contexts are matched, it only needs to consider the same context changes for each fix location. Also, if there is no change with the same context as a fix location, this fix location can be ignored since it means that past patches did not modify such locations. In addition, CCA uses fine-grained changes preserving changed code structures, but normalizing user-defined names. Hence change concretization can be simply done by replacing normalized names with concrete names available in buggy code. We evaluated CCA's effectiveness with over 54K unique collected changes (221K in total) from about 5K human-written patches. Results show that using contexts, CCA correctly found 90.1% of the changes required for test set patches, while fewer than 5% of the changes were found without contexts. We discovered that collecting more changes is only helpful if it is supported by contexts for effective search space navigation. In addition, CCA repair model found 44-70% of the actual fix locations of Defects4j patches more quickly compared to using SBFL techniques only. We also found that about 48% of the patches can be fully concretized using concrete names from buggy code.

**Keywords** Automatic program repair · Context-based change application · Repair models

Communicated by: Martin Monperrus

✉ Jindae Kim  
jdkim@cse.ust.hk

Extended author information available on the last page of the article.

## 1 Introduction

An Automatic Program Repair (APR) technique is a method to fix a given bug by modifying program behavior. Usually, a bug is given in source code form, accompanied by a test suite which verifies the code and reveals faulty behavior of the program. Then the goal of the APR technique is to alter the program behavior to pass all given test cases by modifying the given source code. Many APR techniques with various repair models have been introduced to achieve this goal more effectively (Le Goues et al. 2012; Weimer et al. 2009, 2013; Qi et al. 2013, 2014, 2015; Kim et al. 2013; Long and Rinard 2015, 2016b; DeMarco et al. 2014; Xin and Reiss 2017; Long et al. 2017; Le et al. 2016; Saha et al. 2017; Wen et al. 2018; Jiang et al. 2018; Liu et al. 2019).

The patch generation of APR techniques is often described as an exploration of their patch search space (Martinez and Monperrus 2015; Long et al. 2017; Long and Rinard 2016a; Jiang et al. 2018). In the search space of a technique, each point represents a patch candidate which can be generated by the technique. Then the technique navigates through the space by generating patch candidates until it finds a candidate passing all given test cases. To generate a patch candidate, an APR technique should decide which locations of a buggy program need to be modified (*fix location selection*) and how to modify such locations (*changes selection* and *change concretization*). For instance, GenProg (Le Goues et al. 2012) uses a Fault Localization (FL) technique to choose a suspicious statement as a fix location (fix location selection). Then it selects one of its mutation operators to insert, delete or replace a statement at the fix location (change selection). If insert operator is selected, GenProg selects one of the statements in buggy code and inserts it to the fix location to concretize the operator (change concretization). Hence GenProg explores its search space with such selections required to generate patch candidates.

To fix more bugs correctly, an APR technique needs to satisfy two conditions: large search space and effective navigation. Large search space indicates that various patch candidates can be generated by a technique, hence it is more likely that a correct patch for a given bug is found from the large search space. However, large search space also means that it is difficult to find a correct patch among all those candidates, since only some of them can fix a given bug (Long and Rinard 2016a). Hence effective navigation is required to generate a patch within limited time and resources.

Recent studies and APR techniques tried to solve these issues by expanding search space using existing patches and employing various navigation strategies. Changes collected from past patches provide more variations for patch candidate generation, hence they greatly expand search space compared to previous techniques using a limited number of mutation operators (Le Goues et al. 2012; Weimer et al. 2013) or fix templates (Kim et al. 2013; Long and Rinard 2015, 2016b). To navigate expanded search space effectively, repair models employ various strategies using change's frequency as its probability of selection (Nguyen et al. 2013; Martinez and Monperrus 2015), inferring small enough search space using integer linear programming (Long et al. 2017), or collecting high-level changes represented by change and changed entity types only, and concretizing them with exiting code fragment from buggy code (Wen et al. 2018; Jiang et al. 2018). However, the idea of expanding search space by collecting fine-grained Abstract Syntax Tree (AST) changes preserving changed code structure, then controlling exploration of the space using change's context has not been fully investigated yet.

In this study, we investigate the effectiveness of a repair model employing *Context-based Change Application (CCA)* technique to implement such idea. CCA collects abstract subtree

changes and their AST contexts, and applies them to fix locations only if their contexts are matched. The intuition behind CCA is that an APR technique can mimic existing changes by considering syntactic contexts (i.e., AST contexts). For instance, consider the following identical changes with different contexts.

```
//Context 1
+   if(t != null)
+       t.add(x);
//Context 2
+   T t = new T();
+   if(t != null)
+       t.add(x);
```

The same null checker insertion is applied in two different contexts. The first case is meaningful and prevents an error if `t` is null, but the second case is not meaningful. It is more likely that such null checker insertion is observed frequently in the first context, but not in the second context. Hence if we collect changes with their contexts together and consider the contexts during change application, we can prevent the second case, and encourage the first case. Also, to increase reusability of changes, CCA normalizes user-defined variables, types and methods in changes, and concretizes them using concrete names available at fix locations. For instance, when CCA applies `insert if(var0 != null)`, it replaces abstract variable `var0` with concrete variable `t` available at the fix location in this example. Such adaptation would be also done by human developers if they tried to apply the same change to this location.

CCA repair model can address both search space expansion and navigation issues. Since it uses change representation which preserves structure of code fragments, collected changes can introduce new code fragments for patches. Previous APR techniques often represent a change by change type and changed entity type (e.g, `insert assignment`), and the actual code fragment for the change is obtained from buggy code (Le Goues et al. 2012; Weimer et al. 2013; Wen et al. 2018). On the other hand, CCA also provides the structure of the fragment which might not be included in buggy code, and it can be adapted to buggy code by using variables, types and methods from the buggy code. Hence such collected changes greatly expand search space, compared to the search space limited by code fragments from buggy code.

Then CCA repair model exploits CCA in various ways for effective navigation of such expanded search space. First, CCA is helpful to select a change which will be applied to a fix location, since it only considers changes with the same context among all collected changes. CCA can also be useful to filter out undesirable fix location candidates which have not applicable changes in collected changes, since it means that such candidates have not been modified in past patches. Also, CCA's fine-grained change representation makes change concretization easier compared to more abstract representations. By using such strategies, CCA repair model only needs to explore small search areas defined by filtered fix location candidates and their applicable changes, instead of wandering over the whole search space.

We first evaluated whether using contexts can be helpful to effectively select necessary changes for patches. For evaluation, we collected 54,668 unique AST subtree changes (221,062 changes in total) from 4,995 human-written patches of 11 Java software projects. We also prepared a test set of patches by randomly selecting 220 patches (20 for each project) from the collected patches. For each change in the test set patches, we evaluated whether a change selection method can find the change among all collected changes based on contexts. In a best-case scenario, fewer than 5% of the changes were correctly found

without context, while 90.1% of the changes were found using context. This is significant improvement and clearly indicates that using contexts is effective in change selection.

One major purpose of collecting changes is expanding search space to generate more and various patches. Hence we need to verify that we can cover more changes required for patches when we collect more changes. For this experiment, we increased the size of a change pool which contained collected changes, by gradually adding total 4,775 collected patches to the change pool except for 220 test set patches. As the change pool size (i.e., included patches) increased, more changes included in the test set (test set changes) were covered by changes in the change pool.

However, there exists trade-off in increasing the change pool size. As the size of the change pool was increased, the difficulty of finding a certain change among all the changes in the change pool was also increased. Due to such trade-off, although more test set changes were included as the size of change pool grew, the number of test set changes we could find from the change pool did not show statistically significant difference. On the other hand, we found that we could find significantly more test set changes when we selected changes based on contexts, hence we can actually benefit from more collected changes only if we use contexts.

We also investigated usefulness of contexts in fix location selection. APR techniques usually use Spectrum-Based Fault Localization (SBFL) techniques to obtain a list of fix location candidates (Le Goues et al. 2012; Long et al. 2017; Wen et al. 2018; Jiang et al. 2018; Liu et al. 2019). Then APR techniques modify these candidates to generate various patch candidates and explore search space. Hence, it is important to check fewer fix location candidates for effective navigation. For the experiments, we used Defects4j dataset providing 395 bugs and their patches (Just et al. 2014) with their coverage information (Pearson et al. 2017) for fault localization experiments. We found that fix location filtering with contexts can reduce fix location candidates by more than 60%, while only three patches cannot be generated because CCA removed the correct fix locations. In addition, using contexts and their frequency can be also helpful to prioritize fix location candidates. The frequency of a context is a sum of change frequencies appeared in the context. Hence if a fix location candidate's context has high frequency, it means that such location has been frequently modified by past patches. CCA repair model re-orders fix location candidates with the same SBFL scores in descending order of their frequency, and it ranked 44-70% of the actual fix locations higher than SBFL techniques.

In CCA, change concretization is done by collecting concrete names from buggy code and assigning them to normalized names in each change. We analyzed whether such change concretization is effective on patch generation. For all changes of Defects4j patches, we normalized user-defined variables, types and methods, and recorded their original names. Then we counted how many patches can be concretized using existing concrete names from buggy code. We found that about 48% of the patches can be concretized by concrete names available in buggy code. This is several times higher than previous result, which shows that 11% of code lines in commits can be obtained from existing code (Barr et al. 2014). Hence it is more probable that we may find concrete names required to patch generation from buggy code, than obtaining exact code fragments for patches from buggy code.

Also, we discovered that considering type-compatibility can reduce candidate concrete names for each normalized name and make name selection more easier. For instance, if a normalized variable was originally an integer type, then it can be only replaced with another concrete integer type variable considering type-compatibility. Hence candidate concrete names are reduced, and the probability to select correct names is increased in this way.

In our experiments with Defects4j patches, name selections considering type-compatibility concretized at least 40 more patches than a baseline random name selection method. This result indicates that type-compatibility is an effective guidance for name selection, similar to contexts for change selection.

The followings are major findings of this study.

- We analyze the effectiveness of using change’s contexts in change selection. We found that change contexts are effective to find necessary changes from patch search space expanded by many collected changes.
- We also inspect the usefulness of contexts in fix location selection. Our results show that we can save significant effort to examine fix location candidates by filtering and prioritizing locations based on contexts.
- We evaluate change concretization with name selection. We found that we can find concrete names from buggy code for more patches compared to finding whole code fragments, and leveraging type-compatibility is beneficial for change concretization.

The rest of this paper is written in the following order. First we explain CCA technique and how we can generate a patch candidate with CCA in Section 2. Then we provide research questions and our experiment design to answer the questions (Section 3). After that, we present results with detailed analysis in Section 4. We discuss about threats to validity (Section 5) and related work (Section 6) of this study, and conclude in Section 7.

## 2 Context-based Change Application Technique

### 2.1 Change Collection

The goal of change collection step is building a change pool containing abstract AST changes categorized by their contexts. CCA extracts and normalizes AST changes from human-written source code patches to prepare them for future patch generation in this step.

#### 2.1.1 Change Extraction and Normalization

Figure 1 is an example source code patch represented in unified diff format. There are two code changes in the example - one is adding a null checker and the other is updating

```
//Change 1
public void updateLabel(Node n){
    n = findNode(id);
- n.setLabel(label);
+ if(n != null) {
+     n.setLabel(label);
+ }
//Change 2
- sb.append(nodeType);
+ sb.append(nodeLabel);
```

**Fig. 1** An example source code patch

a method argument. The first change, a null checker insertion is represented by deleting the old method call `n.setLabel(label)` and inserting an if statement containing the same method call in it. The second change is updating a method argument `nodeType` to `nodeLabel`, which is shown as the method call is replaced.

Such code-diff change representation is not suitable for re-use in patch generation because of two reasons. Firstly, it is highly dependent to coding style such as brackets or white spaces, and sometimes changed parts are not clearly expressed. For instance, consider the first change of Fig. 1. If there is another change with different bracket positions or whitespaces (e.g., `n!=null` with no space), then this new change might be considered as a different change due to the differences which do not affect program's behavior. Also, like the second change, it does not precisely identify the changed part (i.e., method argument) from the entire method call. Secondly, these changes include user-defined names, such as `nodeType` or `findNode`, which are probably not available in different programs. Hence reusability of these changes can be low unless such user-defined names are accidentally matched to names used in buggy code that we actually want to modify.

CCA mitigates these issues by collecting abstract AST changes. First, CCA applies a source code differencing technique (Kim and Kim 2016) to code changes and obtains polished AST subtree changes which are in more generally applicable form. AST subtree changes are free from coding style differences, and they also point out changed parts more precisely. For instance, Fig. 2a shows string representation of two AST subtree changes obtained from the two code changes in Fig. 1. In the example (Fig. 2a), the null checker insertion in the code change is represented by a replacement of a method call with a null checker. For the second code change, the corresponding AST change exactly points out that the method argument `nodeType` is updated to `nodeLabel`. Since these changes are actually expressed by changed AST subtrees, this change representation can identify the same change with the same changed code fragments regardless of coding style or whitespaces. Also, like the second change, it identifies that the method argument is updated and removes unnecessary, unchanged parts included in the code-diff representation in Fig. 1.

We consider five different change types in CCA.

- **insert** *t*: insert a subtree *t*.
- **delete** *t*: delete a subtree *t*.
- **move** *t*: move a subtree *t*.

```
//Change 1
replace
  n.setLabel(label);
with
  if(n != null) {
    n.setLabel(label);
  }
//Change 2
update
  nodeType
to
  nodeLabel
```

(a) Extracted AST Subtree Changes.

```
//Change 1
replace
  var0.method0(var1);
with
  if(var0 != null) {
    var0.method0(var1);
  }
//Change 2
update
  var0
to
  var0
```

(b) Normalized Changes.

Fig. 2 Extracted and normalized changes from the source code patch

- **update**  $n_1, n_2$ : update node  $n_1$  to  $n_2$ .
- **replace**  $t_1$  with  $t_2$ : replace subtree  $t_1$  with  $t_2$ .

These AST subtree changes (Fig. 2a) are still not prepared for re-use, since they include user-defined names which are probably not available at different locations. Hence CCA normalizes the changes to obtain abstract AST subtree changes (Fig. 2b). All user-defined variables, types and methods are replaced with abstract names. For instance, variable `nodeType` in the second change is replaced with an abstract name `var0`. Such abstract names will be replaced with concrete variables available at new change locations during change concretization. Variables, types and methods defined in Java Standard Library (JSL) and literal values (e.g., `ArrayList`, `equals()`, `-1`) will not be replaced, since they are more likely available in other programs written in Java.

Note that abstract names are only consistent in each AST subtree. It means that the same concrete name will be replaced with the same abstract name only inside each subtree. For example, both `nodeType` and `nodeLabel` in the second change (Fig. 2a) are replaced with the same abstract name `var0` (Fig. 2b), since they belong to an old subtree and a new subtree respectively. For replace or update changes, there are two subtrees in a change, one is an old subtree and the other is a new subtree. When a replace or update type change is applied to another location, abstract names in the old subtree will not be replaced with collected concrete names, since the old subtree just represents existing code. Hence normalization of the old subtree is only necessary to verify identical changes regardless of concrete user-defined names.

Such normalization method provides an advantage of generating more various changes with the same set of abstract changes, compared to consistent abstract names in both AST subtrees. For instance, consider a concrete change `replace a - b > 0` with `b - a > 0`. It would be converted to an abstract change `replace var0 - var1 > 0` with `var1 - var0 > 0`, if we kept abstract names consistent in both subtrees. Now suppose we are trying to replace `i - j > 0` by applying this abstract replacement. We can only replace the expression to another concrete expression `j - i > 0` based on name consistency. However, sometimes we may need to replace the same expression with `i - k > 0` and this cannot be done with the current abstract change. On the other hand, if we only consider name consistency in the new subtree, we can reproduce both expressions. Therefore, this way of normalization allows us to generate more concrete changes with a smaller number of abstract changes.

While normalizing changes, CCA collects the type information of normalized names to support change concretization. For example, in the first abstract change, `method0` is called by variable `var0` of a user-defined type, and has one `String` type parameter. Such method signature will be helpful to find an appropriate concrete method to replace `method0` to avoid compilation errors.

## 2.1.2 Change Context Identification

After abstract AST subtree changes are obtained from source code patches, CCA identifies AST contexts of the changes. An AST context of a change is represented by the information (e.g., node type or subtree structure) of nodes close to the changed subtree. Such contexts distinguish changes applied in different circumstances, and show that which change is more appropriate and popularly used under a certain situation.

Figure 3 shows two identical changes applied in two different contexts. The null checker will be useful to avoid a Null Pointer Exception (NPE) in the first context, if

```

public void method() { //Parent
    .....
    //Change 1
        n = tree.getNode(id); //Left
-   label = n.getLabel();
+   if(n != null)
+       label = n.getLabel();
        sb.append(label); //Right
    //Change 2
        n = new Node(type); //Left
-   label = n.getLabel();
+   if(n != null)
+       label = n.getLabel();
        sb.append(label); //Right
    .....
}

```

**Fig. 3** Two identical changes with different contexts

`tree.getNode(id)` returns null. However, the second null checker is meaningless since `n` cannot be null due to the previous class instantiation. The difference between the two changes is not the change itself, but code fragments close to the change, which are represented by nodes close to the changed subtree in ASTs. Hence by comparing contexts of changes and fix locations, we can avoid cases like the second, and thus use limited resources more effectively.

In CCA, a context of a change can be decided by Parent (P), Left (L) and Right (R) nodes of a changed subtree representing a changed code fragment. In source code, node P represents an entity which the changed code fragment belongs, nodes L and R indicate code fragments before and after the changed code fragment respectively. For example, in the first change of Fig. 3, node L and its subtree (a tree rooted at node L) represents the assignment of `n` before the change, and node R and its subtree renders the method call `sb.append(label)`. The change's parent node P is a method declaration, since the change happens in a method body.

For contexts, CCA uses the information of these nodes, which can be represented by either their node type (T) or their node hash (H). With node type information, we can design more abstract context types, and with node hash, we can create more distinctive context types.

In CCA, a node type consists of two parts: one is its AST node type, and the other is its syntactic location.

### Definition 1 Node Type

Let  $c$  be the root of a changed subtree in a change. The node type  $t(n, c)$  of node  $n$  used in context of  $c$  is defined as follows.

$$t(n, c) = \begin{cases} n.type[c.loc], & \text{if } n \text{ is the parent of } c \\ n.type[n.loc], & \text{otherwise} \end{cases}$$

where  $n.loc$  is  $n$ 's syntactic location.



Figure 4 shows an example AST representing the changed method call of the second change in Fig. 1. Three SimpleName nodes are under the same MethodInvocation node, but their syntactic locations shown as rectangular nodes are different. Suppose two changes are modifying `sb` and `nodeType` respectively. If we only used AST node type of their parents to describe the changes' context, we would only recognize that the two changes happened at a method call. However, modifying the caller `sb` and the method argument `nodeType` have different impact on program's behavior. Therefore we need to additionally consider syntactic locations to distinguish such cases.

Note that CCA uses changed node's syntactic location (*c.loc*) for parent nodes. For example, consider the second change of the old example Fig. 1, which is a method argument update. Its parent node is the MethodInvocation node (Fig. 4), which is located at a method body. However, we are more interested in the actual change's syntactic location (i.e., method argument) rather than its parent's syntactic location. Therefore for parent nodes, we use change's syntactic location as its parent node's syntactic location instead. In the example, the changed node *c* represents argument `nodeType`, hence the change's parent has node type MethodInvocation[*args*] by using *c*'s syntactic location.

For node hash, CCA uses Dyck word hashing (Chilowicz et al. 2009) to obtain hash values.

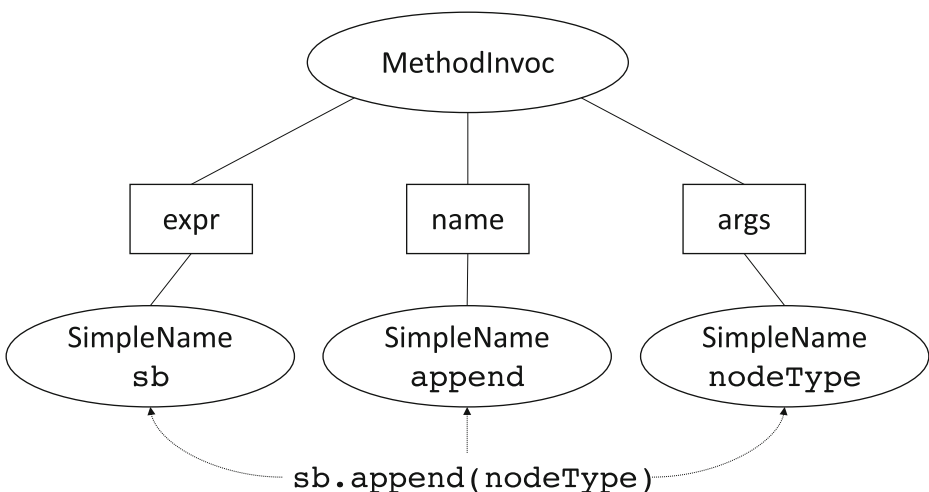
## Definition 2 Node Hash

Node hash  $h(n)$  of node  $n$  with AST node type  $n.type$  is defined as follows.

$$h(n) = \{n.type\{h(c_1), h(c_2), \dots, h(c_k)\}\},$$

where  $c_1, c_2, \dots, c_k$  are child nodes of  $n$

Unlike node type, node hash represents the structure of code fragments as well as their type. For instance, the left nodes of the two changes in Fig. 3 are both assignments. If we use node type, there is no distinction between them. However, if we use node hash, node hash of left nodes for the two changes are {Assignment{SN, MI{SN, SN, SN}}} (Change 1) and {Assignment{SN, CIC{ST, SN}}} (Change 2). In the hash values, SN,



**Fig. 4** An example AST with syntactic locations

MI, CIC, and ST stand for *SimpleName*, *MethodInvocation*, *ClassInstanceCreation*, and *SimpleType* respectively. These node hash values distinguish the two assignments, since they have different right hand side expressions. Hence by using node hash instead of node type, CCA can distinguish contexts more precisely.

There are various possible combinations of nodes with their node type or node hash for contexts. In this study, we consider nine different context types using various nodes and their node type or hash. Figure 5a shows a changed node *C* (the root of a changed subtree) and nodes used for context types, and Fig. 5b provides code fragments correspond to the used nodes. We first consider node *C*'s Parent (*P*), Left (*L*) and Right (*R*) nodes for contexts. For extended contexts, we also use additional nodes which are located at *P*'s ancestors (2*P*, 3*P*), *L*'s left (2*L*) and *R*'s right (2*R*). Node type and hash of these nodes constitute different types of contexts used in this study.

These nodes represent code fragments close to a changed code fragment, or code entity where the changed code belongs in Fig. 5b. For instance, *L* and 2*L* indicate two statements (statement1, statement2) before inserted in the same code block. Similarly, *R* and 2*R* represent two statements after the inserted statement. Ancestor nodes (*P*, 2*P*, 3*P*) designate the location where the change is applied. The statement is inserted in an if statement (*P*), which is under a while loop (2*P*), inside a method (3*P*). These nodes can distinguish change locations more precisely when they are used together. With node type and node hash of these nodes, the context of a change can be defined as follows.

### Definition 3 Change Context

Context *CTX* of a change consists of three components  $C_P$ ,  $C_L$  and  $C_R$ . Let *c* be the root of a changed subtree in a change. Each component  $C_S$  with a set of nodes *S* is a concatenation of node type  $t(n, c)$  or hash  $h(n)$  of node *n* in *S*. Then *CTX* is a string of the form  $CTX = P:C_P, L:C_L, R:C_R$ .

Table 1 shows all nine context types and their components. Note that we represent node type with  $t(n)$  instead of  $t(n, c)$ , since node *c* part is common. From top to bottom, each context type uses more nodes or specific information (i.e., hash instead of type) of nodes. First three context types only use parent nodes or ancestors of the changed node (*parent* group). Context types in parent group are similar to genealogy context considered in CapGen (Wen et al. 2018), which considers a node's ancestors as contexts. The next three context types

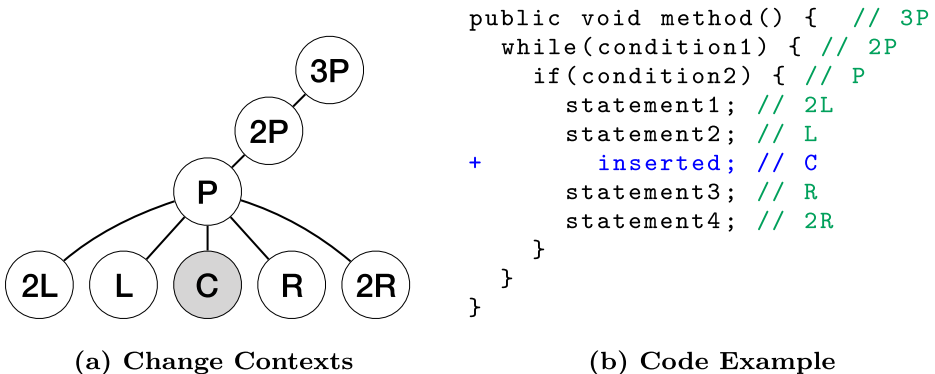


Fig. 5 Node arrangement for change context and code example

**Table 1** Context types and their components

Group	Name	Parent ( $C_P$ )	Left ( $C_L$ )	Right ( $C_R$ )
parent	PT	t(P)		
	2PT	t(P),t(2P)		
	3PT	t(P),t(2P),t(3P)		
type	PLRT	t(P)	t(L)	t(R)
	P2LRT	t(P)	t(L),t(2L)	t(R),t(2R)
	2P2LRT	t(P),t(2P)	t(L),t(2L)	t(R),t(2R)
hash	PTLRH	t(P)	h(L)	h(R)
	PT2LRH	t(P)	h(L),h(2L)	h(R),h(2R)
	2PT2LRH	t(P),t(2P)	h(L),h(2L)	h(R),h(2R)

are represented by nodes' types only (*type* group), which are simple and abstract context types. The last three context types are using node hash for left and right components (*hash* group), which are more precise context types.

Note that h(P) is never used, since a parent's node hash also includes a changed subtree itself. As a result, every unique change has a unique context when we use h(P), and for each context, there will be only one unique applicable change. Therefore, we do not use h(P) for contexts to avoid excessively specific context types.

Different combinations of nodes and information (i.e., node type and hash) affect the resolution of context types. If CCA uses PLRT context, the two changes in Fig. 3 are happened under the same context, since they were applied in the same method body and left, right nodes are both assignments and method calls. However, if PTLRH context is used, the left nodes of the two changes are two different assignments with different code structure, hence the changes belong to different PTLRH contexts. Due to such distinction, using more specific context types works as more strict constraints. Such constraints narrow down search areas which need to be explored during patch generation, among the entire search space.

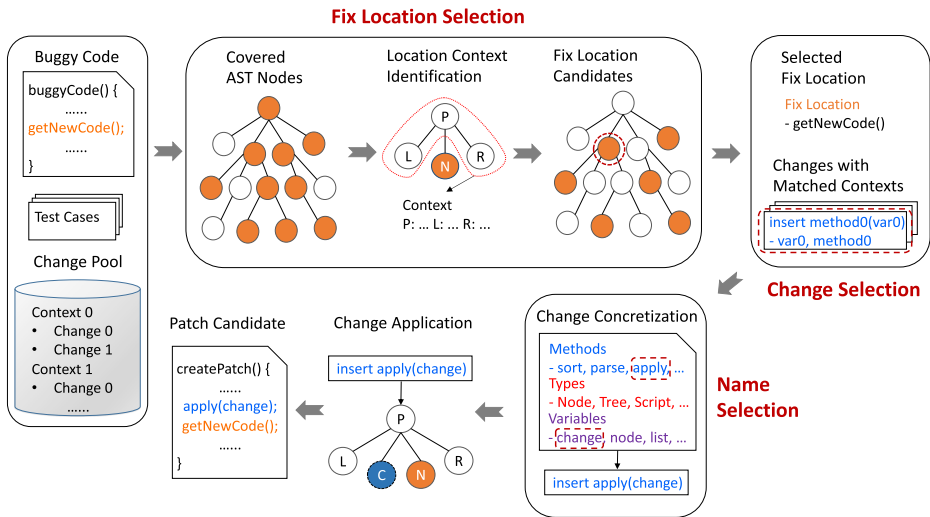
After changes are extracted and their contexts are identified, changes are stored in a change pool and categorized by contexts. Note that move changes are related to two locations since a subtree is moved from one location (old location) to another location (new location). In change pool, move changes are categorized by context identified from its old location, but the context identified in the new location is also stored. Once a change pool is built, CCA repair model can repeatedly use the pool to supply changes to generate patches for many bugs.

## 2.2 Patch Generation

In this section, we explain how CCA uses collected changes and contexts to generate patch candidates.

### 2.2.1 Patch Generation Overview

Figure 6 is an overview of CCA repair model to generate a patch candidate. For a given bug and test cases exposing the bug, CCA repair model generates a patch candidate by applying one of the changes in a change pool collected from human written patches. The first step for patch candidate generation is *Fix Location Selection*. CCA repair model applies a Spectrum-Based Fault Localization (SBFL) technique to identify suspicious statements.



**Fig. 6** The overview of patch generation of a repair model using context-based change application technique

Then it lists up all AST nodes on the statements, and identifies location context of these suspicious nodes. Each location context becomes a possible fix location, where one of the changes with the same context can be applied.

Since CCA only applies changes with the same context to each location, a location with no applicable changes in the change pool can be filtered out. Hence search area for patch generation is further reduced by removing locations which have not been modified in past patches. After that, CCA repair model considers remaining locations as valid fix location candidates, and selects one of them as a fix location of a patch candidate. APR techniques often use SBFL techniques to identify and prioritize fix locations (Le Goues et al. 2012; Long and Rinard 2015; Wen et al. 2018; Jiang et al. 2018). CCA repair model further exploits contexts' frequency, and gives priority to fix locations which were modified more frequently in past patches. A fix location with higher suspiciousness score (SBFL) and higher frequency (contexts) will be selected first.

Once a fix location is selected, it can retrieve a list of changes with the same context from a change pool. The next step is selecting one of the retrieved changes based on a change selection method (*Change Selection*). After that, the repair model collects concrete variables, types and methods available at the selected location. The selected change is concretized by replacing its abstract names with concrete names, which are chosen from the collected names by a name selection method (*Name Selection*). After the change is concretized, it is applied to the fix location and a patch candidate is generated.

### 2.2.2 Fix Location Selection

The key of fix location selection is identifying location contexts from suspicious AST nodes. We design location contexts to make them compatible to change contexts. For each covered AST node *N*, CCA identifies the following four different types of location contexts which can be a candidate for change application.

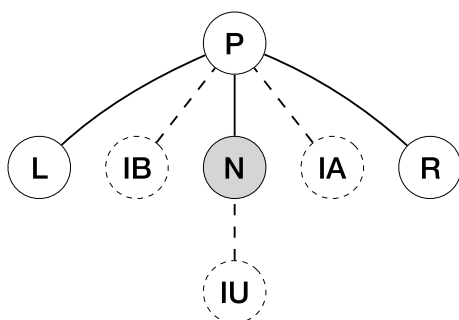
- **Default:** Contexts when node *N* is deleted, replaced or updated.

- **Insert Before:** Assume a subtree is inserted before node  $N$ .
- **Insert After:** Assume a subtree is inserted after node  $N$ .
- **Insert Under:** Assume a subtree is inserted under node  $N$ .

Figure 7a shows node arrangement for location contexts, and Fig. 7b is a code example to illustrate actual code represented by the nodes. In Fig. 7b, the statement `covered` corresponds to node  $N$ , whose location contexts will be identified. In the original code, there are two statements `statementX` and `statementY`, before and after `covered` respectively. When CCA tries to modify node  $N$  by applying delete, replace and update type changes, node  $N$  itself will be changed, hence its Default type context consists of nodes  $P$ ,  $L$ ,  $R$  can be matched to change contexts.

However, when CCA tries to insert a code fragment near `covered`, it requires different types of location context to match them with contexts of insert type changes. In previous change context example (Fig. 5b), `inserted` is inserted between the two statements `statement2` ( $L$ ) and `statement3` ( $R$ ). Now consider cases that `inserted` is actually inserted before (IB) or after (IA) the statement `covered`. For the former case, the actual statement next to the inserted statement is `covered`, hence this should be matched to the right node (i.e., `statement3`) of the change context. For the latter case, `covered` is the statement previous to the inserted statement, and it should be matched to the left node (i.e., `statement2`) of the change context. If CCA used Default type location context and matched it with the change context, it would check whether `statement2` and `statementX` are matched or `statement3` and `statementY` are matched, which are incorrect comparisons. Therefore, CCA needs IB and IA location context types for these two cases, which considers node  $N$  as context's right and left nodes respectively. Note that  $2P$ ,  $3P$ ,  $2L$  and  $2R$  nodes are also decided similar to change contexts (e.g.,  $2L$  is a left node of node  $L$ ), although they are not shown in the example.

Insert Under (IU) context is an additional context type to handle a case that there exist no other nodes to obtain IB or IA contexts. In Fig. 7b, the while statement represents this case. The loop body of the while statement is empty, hence CCA cannot apply insert changes to add more statements inside the loop body with IB or IA contexts as fix locations. Empty loop body can be happened due to a mistake or previous deletions when a series of changes are applied. Hence it is necessary to use IU location context to add new statements in such cases.



(a) Location Contexts

```
public void method() { // P
    .....
    statementX; // L
+   inserted before // IB
    covered; // N
+   inserted after // IA
    statementY; // R
    .....
    while(condition1) { // N
+       inserted under // IU
    }
}
```

(b) Code Example

Fig. 7 Node arrangement for location context and code example

After location contexts are identified, CCA repair model selects one of them as a fix location where a change will be applied. Since every location context is identified from a suspicious node, it has corresponding suspiciousness score computed by an SBFL technique. Hence CCA repair model can select more suspicious location context first based on these scores.

Furthermore, we can exploit context's frequency to identify more desirable fix locations. A context's frequency is a sum of change frequencies for all collected changes in the context. High frequency of a context means that many human-written patches applied changes to that context. Once an ordered list of suspicious location is given by an SBFL technique, CCA further sorts them based on frequencies. In the list, locations with the same suspiciousness score are sorted in descending order of their frequency. In this way, we can also break ties in SBFL's suspiciousness scores and check more desirable locations first.

### 2.2.3 Change Selection

Once fix location is selected, CCA repair model retrieves applicable changes from a change pool. For Default context, all changes with the same context are retrieved, but for the other types of contexts, only insertions are retrieved. If there are move changes in the retrieved changes, they actually require another fix location where a moved subtree is inserted. However, we cannot know whether an addition fix location is necessary until a change is selected, hence we only consider an additional fix location after a move type change is selected and it is being applied. Any selection methods can be used to select one of the retrieved changes with the same context. In this study, we compare the following three change selection methods and assess which method is more effective for change selection in CCA.

- *Random (RND)*: A baseline method. Randomly selects one of the given changes.
- *Roulette-Wheel Selection (RWS)*: A stochastic method with change frequency as weight.
- *Most Frequent First (MFF)*: Selects the most frequent change among given changes.

Random method literally selects one of the applicable changes randomly. RWS method is a stochastic method using a change's frequency as its probability to be selected. It uses *roulette-wheel selection* (Lipowski and Lipowska 2012) for probabilistic selection. MFF method selects changes in the descending order of their frequencies. Hence the most frequent change with the same context is selected to generate a patch candidate at first. For the next change selection to generate another candidate, MFF method selects the next frequent change into the context.

With RWS and MFF methods, CCA repair model can exploit the natural distribution of changes in each context. For instance, inserting a break statement might be popular in a loop body, but it may also be rare outside any control statements. Hence, for each context, it is a reasonable choice to select more frequent changes in that context first. Also, using a change's frequency as its selection probability is a popular method used in previous studies (Martinez and Monperrus 2015; Nguyen et al. 2013).

### 2.2.4 Change Concretization and Name Selection

After a change is selected for a fix location, CCA concretizes the change to generate concrete patch candidate. Change concretization is a process to assign concrete variables, types and methods to normalized names in the change. Hence, change concretization is a series of name selections eventually.

There are many possible methods to select concrete names for abstract names in a change. In this study, we consider four different methods for name selection.

- *Random (RND)*: A baseline method. Randomly selects one of the concrete names.
- *Type-Compatible (TC)*: Consider type-compatibility of variables, types and methods for assignments.
- *Variable-oriented Fault Localization (VFL)*: Consider relevance of names to a bug, computed by variable-oriented fault localization technique (Kim et al. 2018).
- *Hybrid Method (TC-VFL)*: Consider both type-compatibility and relevance to select concrete names.

Random method randomly assigns available concrete names to abstract names in a change. Concrete names are collected from buggy code to adapt the abstract change to the given bug. Note that concrete variables are only assigned to abstract variables, and it is same for concrete methods and types.

In TC method, CCA considers type-compatibility of abstract variables, types and methods to identify candidate concrete names, and randomly selects one of the candidates. For instance, suppose `var0.method0(var1)` is selected, and `var0` has type `Type0`, `var1` is `String` type, and `method0` returns an integer.

To satisfy type-compatibility and avoid errors, there are several constraints for concrete names. Firstly, `method0` must return an integer. Secondly, `method0` must take one `String` type parameter. Lastly, `method0` must be called by a variable `var0` of a user-defined type.

TC method first lists up all methods satisfying these constraints, and randomly selects one of them. By this method selection, the user-defined type of `var0` is decided. If there exist any other abstract types, TC method randomly selects one of the concrete types for each abstract type. At this point, all abstract types are replaced with concrete types, hence all variable types are also fixed. Finally, TC methods selects one of the concrete variables with the same type for each abstract variable.

VFL method computes each name's score representing its relevance to the given bug. In *Variable-oriented Fault Localization* technique (Kim et al. 2018), variables appeared in statements executed by failing test cases are considered more suspicious, and lines with more suspicious variables are considered more suspicious faulty locations. Suppose a variable was shown in five statements covered by failing tests. Then the probability that this variable is related to the given bug is higher than other variables not shown in any statements covered by failing tests. Hence, using more relevant variables for a selected change is more desirable than using irrelevant variables to fix the given bug.

We compute scores of each concrete variable, type and method name  $n$  like follows.

$$Score(n) = \frac{\sum_l occur(n, l) \times W(l)}{|lines(n)|} \quad (1)$$

In the equation,  $occur(n, l)$  is the number of occurrences of name  $n$  on line  $l$ ,  $|lines(n)|$  indicates the number of lines which  $n$  is appeared, and  $W(l)$  is the weight of the line  $l$ , which is defined as follows.

$$W(l) = \begin{cases} 10, & l \text{ is covered by failing tests.} \\ 2, & l \text{ is covered by passing tests only.} \\ 1, & l \text{ is not covered.} \end{cases} \quad (2)$$

Based on this score, names appeared frequently in lines covered by failing tests will have high scores. Also, the weight for lines only covered by passing tests is still higher than lines

not covered by tests. Hence a name only appeared non-covered lines will have a lower score than a name appeared in covered lines, since the covered names are more likely involved to test execution than non-covered names. To leverage computed scores, VFL-based method uses roulette wheel selection (Lipowski and Lipowska 2012) with the scores as weights.

The hybrid method TC-VFL considers both type-compatibility and relevance of concrete names. First, it identifies candidate concrete names considering type-compatibility, same as TC method. However, instead of randomly selecting one of the candidates, TC-VFL method uses roulette wheel selection with candidates' scores, same as VFL method.

Once change concretization is finished, the selected change is applied to the selected fix location to generate a modified AST representing a patch candidate. Search space exploration can be done by repeating this patch candidate generation until one of the generated candidates passing all given test cases.

### 3 Experimental Setup

#### 3.1 Research Questions and Experiment Design

##### 3.1.1 RQ1: Is CCA Effective on Change Selection?

The key idea of CCA is providing abundant changes for various patch generation, while finding necessary changes effectively based on contexts. To evaluate finding changes part independently, we first collected human-written patches, and randomly selected some of the collected patches to prepare a test set. Then we checked that CCA repair model can find changes used in the test set patches among all changes included in the collected patches.

More specifically, for each change included in a patch, we retrieved changes with the same context from a change pool. Then we applied change selection methods introduced in Section 2.2.3, and checked whether a selected change is matched to the change in the patch. If we assume that a correct fix location for this change is selected, then the location's context is identical to the context of the change applied to it. Hence we can evaluate change selection independently regardless of fix location selection, by using the context of the change included in the patch.

After we repeated such change selection for all changes in the patch, we computed *Matched Change Ratio (MCR)* for a patch.

$$MCR(P) = \frac{\text{Number of Matched Changes in Patch } P}{\text{Total Number of Changes in Patch } P} \quad (3)$$

Note that two changes are matched if their change types (e.g., insert, delete) are identical and changed subtrees are label-isomorphic. The label of a node is represented by its type and value. For instance, the first leaf node of Fig. 4 has label `SimpleName::sb`, and after normalization, it will be `SimpleName::var0` which represents a changed node in an abstract change. Hence two label-isomorphic subtrees indicate two identical code fragments whether they are normalized or not.

MCR represents how much portion of changes can be found by a change selection method, which is similar to *graftability* of a previous study (Barr et al. 2014), representing how many code fragments are found from codebase. We compared average MCR for all patches in the test set for change selection with and without contexts, and among different context types and selection methods.



### 3.1.2 RQ2: Is Collecting More Changes Useful Under Contexts?

Although it is possible to select changes more effectively based on their contexts, it is also possible that too tight context constraints restrict patch generation capability. For instance, suppose we need three unique changes applied to three different contexts to generate a patch. Without contexts, we can generate the patch if collected changes include the three changes. However, when we consider contexts, we also need that changes are collected from the same contexts to generate the same patch. We might need more changes to cover the same amount of changes with contexts, or we might not even cover any more changes although we are considering more collected changes. Therefore, we evaluate whether collecting more changes are actually beneficial to cover more changes required to generate more patches.

For this evaluation, we divided the collected patches into two sets. One is the test set used in the previous evaluation (Section 3.1.1), and the other is a training set, which contains the remaining patches. Then we built an empty change pool, and added changes included in 20 patches randomly selected from the training set. As more patches were added to the change pool, more changes were included and the size of the change pool was increased. Every time we increased the size of the change pool, we checked how many changes used in the test set patches can be covered by the changes in the change pool, when we considered different context types.

### 3.1.3 RQ3: Is CCA Useful to Find Fix Locations?

CCA repair model leverage contexts to filter out undesirable fix locations and prioritizes identified fix locations. We first checked that fix location filtering with contexts successfully reduced fix location candidates without excluding actual fix locations modified in human-written patches. More filtered fix location candidates indicate smaller search area in the search space, hence effort to find a patch could be also reduced. However, if it excluded too many fix location candidates, it might interfere patch generation by removing locations which should be modified.

To assess both aspects, we computed fix location filtering score ( $Score_{FL}$ ), which is a weighted sum of filtered location ratio and covered patch ratio.

$$Score_{FL} = w_1 \times (1 - ValidLoc) + w_2 \times Patches \quad (4)$$

$ValidLoc$  is the number of valid fix location candidates after filtering, and  $Patches$  is the number of patches whose entire fix locations are included in the valid fix location candidates. For weights, we used  $w_1 = 0.5$ ,  $w_2 = 0.5$ , which treated filtering effect and preserving actual fix locations evenly.

Note that CCA repair model also prioritizes fix location candidates based on context frequencies. To assess the effectiveness of context-based fix location prioritization, we first obtained two ordered lists of fix locations, one ordered by an SBFL technique only (FL) and the other ordered by both SBFL and context frequency (FL-FREQ). Then we computed the rank of actual fix locations of the test set patches, and compared the rank differences between the two lists.

### 3.1.4 RQ4: Which Name Selection Method is More Effective?

To complete a concrete patch, it is important to select right concrete names for abstract names in changes. However, this aspect was sometimes neglected in previous studies (Nguyen et al. 2013; Martinez and Monperrus 2015), which only compared normalized

**Table 2** Human-written patches and unique changes collected from the patches

Project	Patches	Changes	Freq.	Project	Patches	Changes	Freq.
<i>collections</i>	74	489	852	<i>lang</i>	172	1,014	1,650
<i>derby</i>	934	6,730	12,765	<i>lucene</i>	184	2,890	4,737
<i>groovy</i>	1,250	13,960	63,407	<i>mahout</i>	262	3,471	8,099
<i>hadoop</i>	447	17,574	100,729	<i>math</i>	338	4,279	10,158
<i>hama</i>	63	838	1,114	<i>pdfbox</i>	993	7,752	13,897
<i>ivy</i>	278	2,437	3,654	Total	4,995	54,668	221,062

Freq. column shows the total occurrences of the changes. Total row shows the numbers for the whole set. Note that the number of total unique changes is less than the summation of unique changes of all projects due to common changes across projects

values or simply matched change type and changed entity types. In this study, we evaluated the idea of using collected concrete names from buggy code to concretize abstract changes in two ways. First, we checked how many patches can be concretized by existing concrete names. Next, we evaluated the effectiveness of name selection methods to find correct concrete names.

For this evaluation, we consider two sets of existing concrete names. First, *internal* names are variables, types and methods explicitly appeared in the same Java class modified by a change. Since these names are used in other locations of buggy code, it is more likely that these names are related to a bug. We also consider member variables and methods of imported classes as *external* names. These names are not explicitly used in buggy code, but they are still available and could be related to a bug since they are included in other classes imported by developers. We checked how many changes and patches can be concretized by internal names, and how many more changes and patches can be concretized if we also used external names.

To compare performance of name selection methods, we collected original names of normalized variables, types and methods from Defects4j patches. For each collected change, we used each name selection method (Section 2.2.4) to assign concrete names for abstract names in the change. If an assigned concrete name was identical to the original name, we considered this assigned name was matched to the abstract name. Then we measured the number of concretized changes whose names were completely matched by each name selection method. In case of patches, we considered a patch was successfully concretized if all changes in the patch were concretized.

### 3.2 Data Collection

To conduct experiments for CCA's effectiveness assessment, we need to collect changes and their contexts from human-written patches. Table 2 shows simple statistics of collected human-written patches and abstract subtree changes.

To collect human-written patches, we first listed all issue numbers categorized as bugs from 11 Java software projects, obtained at Apache's issue tracking system.<sup>1</sup> These selected projects implement various functionalities, and collected patch and change numbers are also diverse from low to high. Then we identified 4,995 patch commits containing the collected

<sup>1</sup> Apache's JIRA issue tracker (<https://issues.apache.org/jira>).

issue numbers in their commit logs. Each commit was considered as one human-written patch, and we extracted 54,668 unique changes (221,062 changes in total) and their contexts from each patch to construct change pools with all context types introduced in Table 1.

For name selection and fix location related experiments, we need coverage information of bugs and their patches since these information is used by fault localization and name selection methods. Obtaining coverage information of collected bugs costs a lot of resources. Hence for the two experiments, we used Defects4j dataset, which provides 395 Java bugs and their patches as well as coverage information for fault localization experiments (Just et al. 2014; Pearson et al. 2017), instead of the test set used in change selection experiments.

### 3.3 Change Pools

From collected human-written patches, we built nine change pools with nine different context types introduced in Table 1.

Table 3 shows nine change pools with different context types. Contexts column shows the number of identified contexts for each context type. Context-Unique column indicates the number of distinguished changes. If a unique change appears in several contexts, we consider each of them as one context-unique change. Hence the number of context-unique changes is higher than 54,668 unique changes which is counted without consideration of contexts. Avg. Changes column presents the average number of unique changes per each context. These numbers are related to a context type's practical resolution, since it generally represents how changes are scattered to contexts. Hence we can define the resolution  $Res(CT)$  of context type (CT) as a reciprocal of average changes per context.

$$Res(CT) = (\text{Avg. number of changes per context})^{-1} \quad (5)$$

Since CCA only considers changes with the matching context, average number of unique changes per context indicates average number of changes considered by CCA when a certain context type is used. Although we collected over 54K unique changes, CCA only needs to check less than two changes on average with context types in hash group. CCA considers about 420 changes even for PT context using only parent type, which is only 0.77% of all unique changes.

In addition to reduced number of changes, CCA can also exploit local frequency distribution of changes. Since CCA scatters collected changes into various contexts, and only counts occurrences in each context, less frequent changes in the entire change set will have

**Table 3** Change pools built from 54,668 unique changes

Type	Contexts	Context-Unique	Avg. Changes	Res(CT)
PT	131	55,053	420.25	0.002
2PT	1,129	59,775	52.95	0.019
3PT	3,376	63,320	18.76	0.053
PLRT	2,281	66,092	28.98	0.035
P2LRT	8,855	73,883	8.34	0.120
2P2LRT	17,000	78,204	4.60	0.217
PTLRH	45,924	81,363	1.77	0.565
PT2LRH	61,574	90,678	1.47	0.680
2PT2LRH	70,725	94,063	1.33	0.752

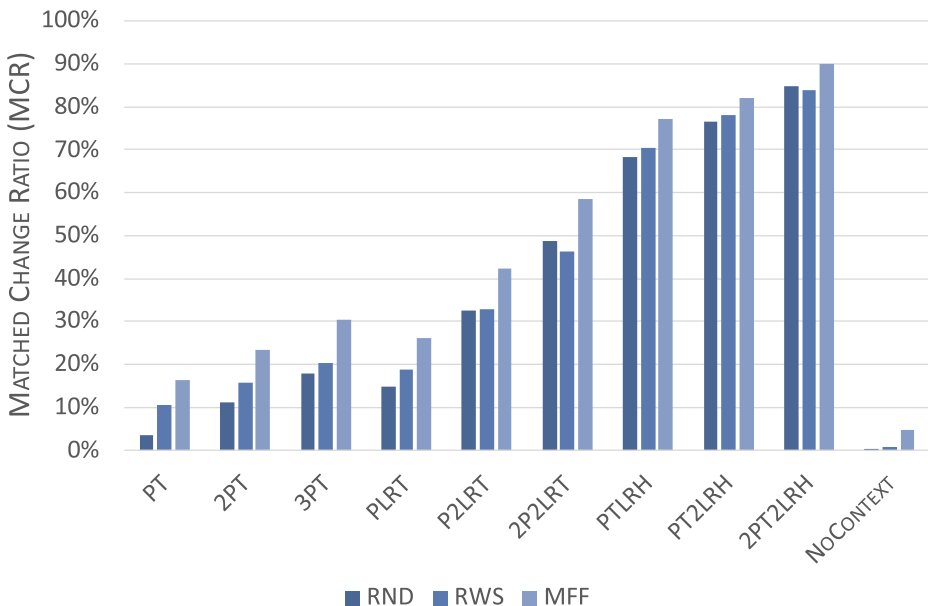
fair chance to be selected in a certain context. For instance, if we consider the entire collected changes, the most frequent change is updating a variable name. However, if we check changes applied right before a return statement under an if statement, the most frequent change is inserting a method call `var0.method0(var1)`. This insertion is only the 82nd most frequent change among the all changes, which would have much smaller probability to be selected if we used frequency distribution for the entire change set.

## 4 Evaluation Results and Discussion

### 4.1 RQ1: Effectiveness on Change Selection

To evaluate CCA repair model's effectiveness on change selection, we computed MCR of RND, RWS and MFF methods for 220 patches with 3,408 changes. Since RND and RWS methods are nondeterministic, we repeated each change selection experiment 20 times, and computed the average MCR of the 20 results for each patch.

Figure 8 shows average MCR of RND, RWS and MFF change selection methods with top-10 choices. For each actual fix location of a patch, we used each change selection method to obtain top-10 choices of changes, and checked whether one of the selected changes was matched to the actual change applied to the given fix location. We used top-10 choices since generate-and-validate techniques often generate many different patch candidates by modifying the same fix location more than once (Le Goues et al. 2012; Kim et al. 2013; Wen et al. 2018; Long and Rinard 2015; Jiang et al. 2018), hence checking top-10 choices similarly simulates such techniques. MCR was computed based on these change selections, and we computed MCR for each patch, and reported average MCR of all patches for each change selection method.

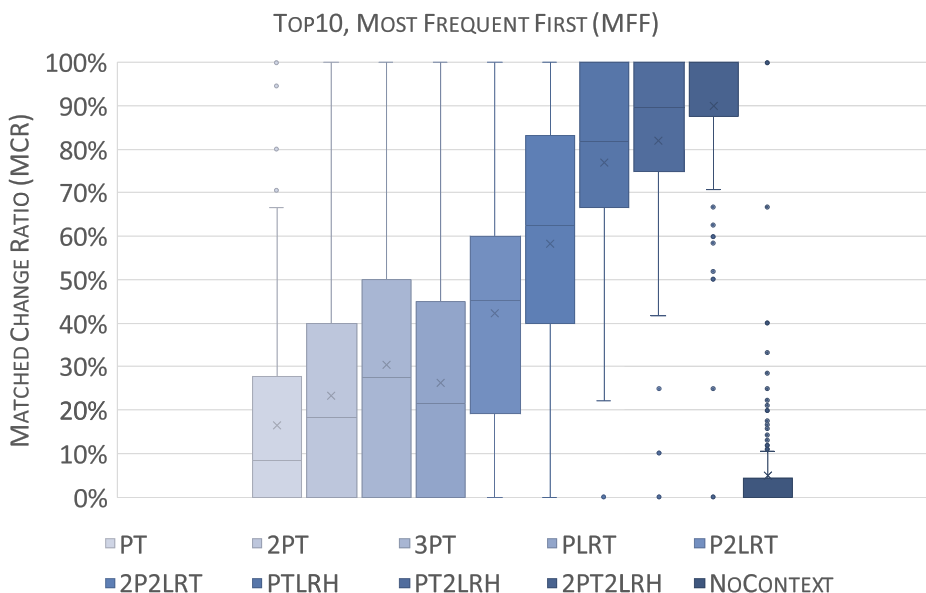


**Fig. 8** Average MCR of top-10 changes with RND, RWS and MFF methods

For change selection performance, using any of the context types significantly outperforms no context. Without context, average MCR is less than 5% (RND:0.02%, RWS:0.66%, MFF:4.8%), while using context types show 3.51–90.08%. With context, using PT context with RND method shows the lowest MCR (3.51%), but it is more than a hundred times higher than RND without context. For the best case (2PT2LRH, MFF method), average MCR reaches 90%, which indicates that MFF method can find 90% of the changes required for patches on average. This MCR is about 18 times of MFF method without context, which shows that using context significantly improves change selection performance.

In terms of change selection method, MFF method outperforms the other methods for all context types including no context. Both RWS and MFF methods consider change's frequency. The key difference is that MFF method guarantees that high frequency changes are selected first, while RWS provides higher probability to more frequent changes. This result indicates that the loss of not choosing high frequency changes is surely higher than the gain of providing chance to low frequency changes. It is possible that many patches consist of several frequent changes and a few low frequency changes. By ensuring high frequency change matches, average MCR itself is higher, but it might not be helpful for patch generation. However, we also observed that MFF method outperforms with all context types in the number of fully matched patches (i.e., patches with 100% MCR). For instance, MFF method fully matched 52.72% of the patches with 2PT2LRH context in the best case, while RWS method matched 41.39% with the same context type, which is even lower than RND method (43.23%). Therefore under context constraints, using MFF method is certainly more effective than the other methods.

Figure 9 shows MCR distribution of 220 patches with 3,408 changes, computed by top-10 selected changes with MFF method. Since MFF method outperforms the other methods, we only posted the result of MFF method to evaluate the influence of context types more



**Fig. 9** MCR distribution of top-10, most frequent first

closely. For each box plot, a line in the middle of box indicates the median, and the cross mark presents the average. There are 10 different context types, including “NoContext” which indicates selecting changes without contexts. First nine context types are placed in the order of parent, type and hash group (three types for each group) shown in Table 1. In each group, three context types are presented in ascending order of their resolution.

The result shows that high resolution context types tend to show better performance in change selection. High resolution means that there are fewer number of changes in each context, hence selecting one of them within 10 trials is easier than low resolution. For instance, only first four context types in Table 1 have more than 10 changes on average for each context. Hence for the other context types, top-10 choices usually contain most of the changes in the same context, which improves MCR.

However, the improved performance is not entirely caused by reduced number of choices, since we used MFF method. For instance, suppose there was a change with frequency 10 in a certain context. With higher resolution context, this change might be applied in two different contexts, with frequency 6 and 4 respectively. If there was another change with frequency 7, and this change was still applied to the same higher resolution context, it was considered more frequent than the first change in the higher resolution context. Conversely, the first change’s frequency became lower, hence it might not be selected by MFF method due to reduced frequency. However, even with MFF method, high resolution context types show improved performance, which indicates that using *local change frequency* - change frequency in each context - can be effective for change selection in addition to reduced number of choices.

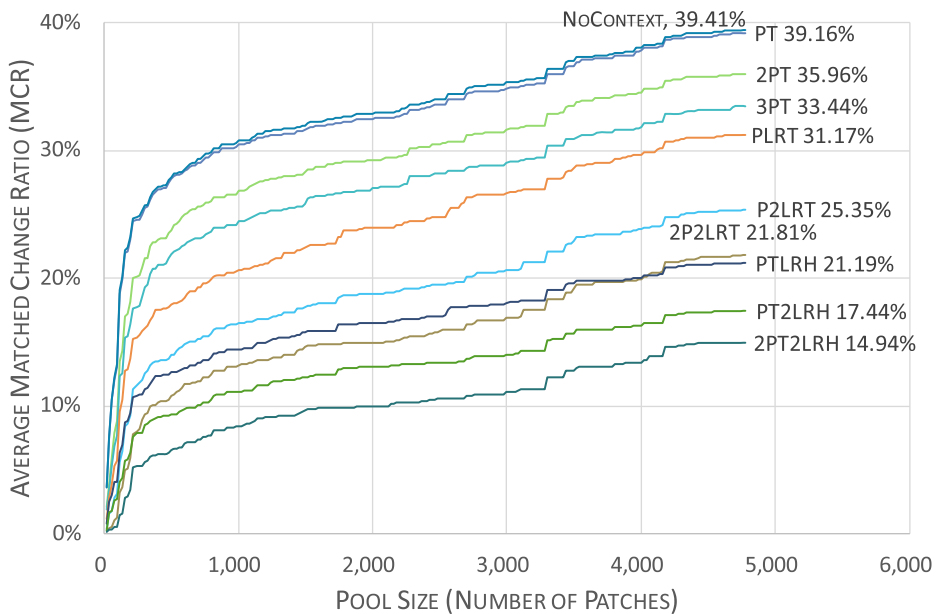
We also checked patches fully matched by MFF method for each context type. Without context, there is only one fully matched patches, which is 0.46% (1/220) of the test set. With PT to 2PT2LRH contexts, 1.36% to 52.72% patches are fully matched by MFF method. In Fig. 9, box plots for hash group contexts biased toward to the top, which indicates that there are more fully matched patches compared to other context types showing only whisker or a single dot (NoContext).

- *MFF method is the most effective change selection method in all cases.*
- *Using any context types significantly improves change selection performance, and higher resolution context type provides better change selection performance.*
- *Local change frequency in each context is also useful in addition to reduced number of changes for selection.*

## 4.2 RQ2: Is Collecting More Changes Useful Under Contexts?

The previous result shows that using context is much more effective in change selection compared to no context. However, this result is only meaningful if necessary changes are included in patch search space. The main reason of collecting changes from human-written patches is expanding the search space to include correct patches for more bugs. Therefore, we assessed whether collecting more changes was actually helpful to increase MCR under different context types.

Figure 10 shows how average MCR of different context types increased as we added more patches to a change pool. To obtain average MCR, we first computed MCR for each patch, and computed the average of MCRs of all the test set patches every time we added more patches to the change pool. Note that to eliminate the influence of change selection



**Fig. 10** Average MCR of change pool with different number of collected patches

method, we checked every changes in the same context for each change included in the test set patches.

Overall, as more patches are added to the change pool, more portion of the test set patches are covered by changes in the added patches. After rapid increment before 1,000 patches, average MCRs of all context types gradually increase in similar rate until all collected patches are added to the change pool. NoContext reaches 39.41% MCR, which means that about 40% of changes in test set patches can be found from other patches on average. Under contexts, average MCR varies from 14.94% to 39.16%. With PT context, the lowest resolution type, average MCR is similar to no context. Since there are only 131 different PT contexts (Table 1), most of the changes in test set patches can be found in the same PT context. However, as higher resolution context types are used, the same set of changes are scattered to more various contexts, hence satisfying the same context condition becomes more difficult.

This result indicates that high resolution context types have disadvantage in leveraging collected changes. If we use 2PT2LRH contexts, we can only obtain about 15% of the necessary changes from other patches on average, but we can expect 40% of the necessary changes if we ignore contexts. It means that we need to collect more patches under contexts to expand patch search space, compared to no context. For instance, in Fig. 10, PLRT context reaches 30% MCR with 4,000 patches, while NoContext achieves it within 1,000 patches.

However, the necessity of more patches does not mean that using context is not useful. In the previous change selection results, we observed that finding necessary changes in expanded search space is very difficult without contexts. Expanding search space is only meaningful if we have an effective method to find necessary changes. On the other hand, collecting more patches is relatively easy, compared to designing an effective change selection method without context.

In terms of patches with 100% MCR, there are 6 (2PT2LRH) to 14 (PT, No context) fully covered patches. However, these low numbers of the fully covered patches do not mean that CCA is not suitable for patch generation. Although we tried to obtain commits related to bug-fixes, software commits are usually composite and address multiple issues at the same time (Tao et al. 2012; Tao and Kim 2015). Hence it is possible that a bug can be fixed even if CCA cannot achieve 100% MCR for the bug's patch.

We found that 131 patches contain method declarations, type declarations or specific string literals (e.g., error messages), which are difficult to be obtained from other code base. Since CCA leverages change repetitiveness, it works effectively for small, frequent changes. However it is less effective for large changes such as inserting method or type declarations. If any of such changes are not found from outside resources, 100% MCR cannot be achieved. When we ignored changes related to these three entity types and counted fully covered patches, there were 10 (2PT2LRH) to 31 (PT), 32 (No Context) fully covered patches, which were almost twice of the original results.

To further analyze relations between expanded search space and change selection using contexts, we checked whether more changes can be found by MFF method as more patches are collected. As more patches are added to a change pool, the probability that necessary changes are included in the change pool is increased, but the probability that those changes are found is decreased. Hence we need to test whether we can actually find more changes for patches if we provided more collected patches under such trade-off.

We used Wilcoxon signed rank test on MCR of the test set patches for the baseline 500 size change pool and other increased size change pools, to test the following hypotheses for 95% confidence interval.

- $H_0$ : Increasing pool size does not affect MCR of patches.
- $H_1$ : Increasing pool size affects MCR of patches.

The null hypothesis ( $H_0$ ) indicates that adding more collected patches to a change pool is not helpful to find more changes used in patches. The alternative hypothesis ( $H_1$ ) represents the case that MCR is actually affected with more patches. We tested the above for all 10 context types with increased size change pools.

Table 4 shows p-values of each context type for 1,000 to 4,775 size change pools. Bold text means that the p-values are less than 0.05, hence we take the alternative hypothesis which indicates that collecting more patches affected MCR significantly. For such significant cases, '↑' means that MFF method tends to find more necessary changes as more patches are added, and '↓' indicates the opposite.

We found that collecting more patches are not always useful to find more changes required for the test set patches. For instance, with PLRT context, MFF method found significantly more changes for the test set patches when first 1,000 patches were added to the change pool. However, when more than 1,000 patches were added to the change pool, there was no significant change in MCR. Note that average MCR kept increased in Fig. 10 as more patches were collected. Although more collected patches supplied more changes required in the test set, they also increased the difficulty of finding them, hence there was no significant improvement. For NoContext, PT and 2PT context types, MCR was sometimes even significantly decreased (bold text with ↓) after more patches were added. On the other hand, high resolution context types (bottom four types in Table 4) continuously show significantly increased MCR after enough patches were added to the change pool. Therefore, collecting more patches is helpful to find changes required in patches only if change selection is supported by high resolution contexts.



**Table 4** The result of Wilcoxon signed rank test for MCR

Context	Size (Number of Patches)				
	1,000	2,000	3,000	4,000	4,775
NoContext	0.2945	0.2086	0.4873	<b>0.0015</b> ↓	0.2086
PT	0.1416	<b>0.0000</b> ↓	0.2278	0.7998	0.5468
2PT	0.2710	<b>0.0014</b> ↓	0.1839	0.9284	0.1850
3PT	0.7502	0.0781	0.8289	0.0627	0.9452
PLRT	<b>0.0364</b> ↑	0.3610	0.4026	0.9613	0.8554
P2LRT	0.9367	0.1423	0.6344	0.2211	0.8208
2P2LRT	0.3674	0.2365	<b>0.0057</b> ↑	<b>0.0000</b> ↑	<b>0.0018</b> ↑
PTLRH	0.2628	0.1312	0.1749	<b>0.0022</b> ↑	<b>0.0023</b> ↑
PT2LRH	0.0515	0.5325	0.1998	<b>0.0009</b> ↑	<b>0.0121</b> ↑
2PT2LRH	<b>0.0004</b> ↑	<b>0.0002</b> ↑	<b>0.0000</b> ↑	<b>0.0000</b> ↑	<b>0.0000</b> ↑

- *Collecting more patches can supply more changes required in patches for all context types, but high resolution context types require more collected patches to achieve the same MCR.*
- *Collecting more patches is helpful to find necessary changes only with high resolution contexts.*

### 4.3 RQ3: Is CCA Useful to Find Fix Locations?

Table 5 shows fix location filtering results for 395 Defects4j patches. For each row of context type, Valid Loc shows the ratio of valid fix locations candidates after filtering based on contexts. For instance, with PTLRH context, we only need to check 50.47% of the all possible fix locations covered by failing test cases. Patches column represents the number of patches whose actual fix locations are completely included in valid fix location candidates. In case of PTLRH context, 62 patches can be generated by modifying valid locations, but the other patches cannot be generated since some of their actual fix locations are filtered by contexts.  $Score_{FL}$  is computed by Eq. 4 with Valid Loc and Patches column values, to summarize both columns.

**Table 5** Fix location filtering results of 395 defects4j patches

Context	Valid Loc	Patches	$Score_{FL}$
NoContext	100.00%	136 (34.43%)	0.17
PT	96.86%	136 (34.43%)	0.19
2PT	43.81%	134 (33.92%)	0.45
3PT	38.44%	133 (33.67%)	0.48
PLRT	65.80%	125 (31.65%)	0.33
P2LRT	38.01%	117 (29.62%)	0.46
2P2LRT	35.33%	109 (27.59%)	0.46
PTLRH	50.47%	62 (15.70%)	0.33
PT2LRH	28.62%	52 (13.16%)	0.42
2PT2LRH	25.91%	46 (11.65%)	0.43

Based on  $Score_{FL}$ , 3PT context is the best type for fix location filtering considering the both aspects. We only need to check 38.44% of all possible fix locations, while sacrificing only three patches. Note that although valid locations include all actual fix locations of a patch, this patch may not be generated within limited time, if too many locations should be examined until the patch is generated. Hence saving effort to check more than 60% of the fix locations is a huge benefit in search space navigation.

In each context group (i.e., parent, type, hash), higher resolution context type filters out more fix locations, but it also decreases the number of patches can be generated. If we consider filtering effect and generated patch numbers equally, higher resolution context types are more effective in each group, as represented in higher  $Score_{FL}$ . In other words, high resolution context types are more effective in search space navigation similar to change selection. By checking smaller search areas in the search space, high resolution context types can reduce effort to find each patch. However, it is also possible that reduced search areas do not include a patch which we are currently finding. To mitigate negative influence of such trade-off, we may consider a repair model sequentially using different context types. For instance, a search for a patch may start with high resolution context, checking small areas first. If the search is unsuccessful, we can extend search areas using low resolution context to find the patch.

We also compared fix location prioritization methods using only an SBFL technique (FL) and using both SBFL and context frequency (FL-FREQ) explained in Section 3.1.3. We first obtained two ordered lists of fix location candidates for all 395 Defects4j patches. Note that FL method orders the same score fix locations in ascending line numbers and depth-first visiting order, since all locations on the same statement have the same suspiciousness score. Then we computed the rank of actual fix locations of the patches, and applied Wilcoxon signed rank test with the following hypothesis to compare the rank differences between the two lists.

- $H_0$ : Fix Location Ranks of FL and FL-FREQ methods are not different.
- $H_1$ : Fix Location Ranks of FL and FL-FREQ methods are different.

Table 6 shows p-values and the ratio of actual fix locations whose rank is going up and down when we use FL-FREQ instead of FL. We computed suspiciousness scores with

**Table 6** The result of Wilcoxon signed rank test on location ranks

Context	Jaccard			Ochiai		
	p-value	Rank Up	Rank Down	p-value	Rank Up	Rank Down
PT	<b>0.0000</b> ↑	0.70	0.24	<b>0.0000</b> ↑	0.70	0.24
2PT	<b>0.0000</b> ↑	0.56	0.36	<b>0.0000</b> ↑	0.55	0.37
3PT	<b>0.0000</b> ↑	0.56	0.35	<b>0.0000</b> ↑	0.55	0.35
PLRT	<b>0.0000</b> ↑	0.64	0.33	<b>0.0000</b> ↑	0.64	0.33
P2LRT	<b>0.0180</b> ↓	0.44	0.52	<b>0.0018</b> ↓	0.43	0.53
2P2LRT	0.2581	0.44	0.50	0.0529	0.43	0.50
PTLRH	<b>0.0001</b> ↑	0.53	0.43	<b>0.0015</b> ↑	0.52	0.44
PT2LRH	0.3518	0.45	0.50	0.1378	0.45	0.50
2PT2LRH	0.9627	0.45	0.48	0.9277	0.45	0.47

Jaccard (Chen et al. 2002) and Ochiai (Meyer et al. 2004) which are popular SBFL techniques in automatic program repair. Bold text ( $p\text{-value} < 0.05$ ) means that we can reject  $H_0$  and accept  $H_1$ , hence the rank difference is statistically significant. Arrows indicate that more actual fix locations are ranked higher ( $\uparrow$ :Rank Up>Rank Down) or lower ( $\downarrow$ :Rank Up<Rank Down) with FL-FREQ method.

Overall, five context types show improved results on fix location prioritization for both Jaccard and Ochiai. For these low resolution context types, the rank difference between FL and FL-FREQ methods are statistically significant, and there are more actual fix locations whose ranks are higher with FL-FREQ method. For instance, FL-FREQ method with PLRT contexts places 64% of the actual fix locations at higher rank than FL method. This means that we need to check fewer number of fix locations until we reach to the correct fix location. It also lowers the rank of 33% of the fix locations, but the advantage is almost twice compared to the disadvantage. Note that high resolution context types have only several changes in each context on average. As a result, context frequency is relatively low and does not vary much between contexts, hence using context frequency to adjust fix location rank is less effective in high resolution contexts.

- *Using high resolution context is more effective to filter out fix location candidates.*
- *However, low resolution context is more helpful in fix location prioritization.*

#### 4.4 RQ4: Which Name Selection Method is More Effective?

In name selection experiments, we identified 1,255 changes from 395 Defects4j patches, and only 1,078 changes from 365 patches include abstract names which require concrete names. Note that some changes such as deletions or updating number literals do not require concretization since they do not have abstract names. For 365 patches with abstract names, we checked how many of them can be concretized using concrete names collected from buggy code.

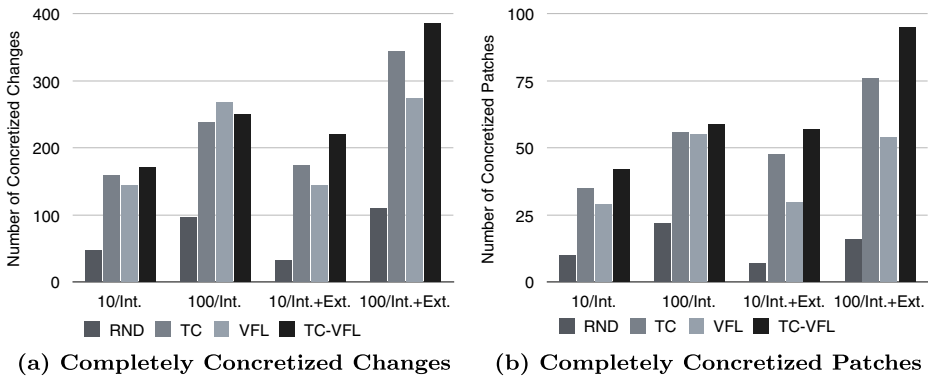
Table 7 shows the number of changes and patches can be concretized by collected names. With internal names, 38.78% of the changes and 31.78% of the patches can be concretized. If we also considered external names, numbers are increased to 51.11% and 47.67% respectively. It means that many of the patches require variables, types or methods which do not appear in modified classes. Hence considering all available concrete names - even if they are not explicitly used in buggy code - can be helpful to generate more patches.

The numbers of concretized patches are several times higher compared to a previous study which considers the whole changed code fragments. Barr et al. found that for 11% of commits are fully *graftable*, which means that we can find all code fragments used in the commits from a code base (Barr et al. 2014). In our evaluation, we found concrete names for each patch completely for 31.78-47.67% of the collected patches. Hence it is more likely that we can find necessary concrete names from a code base rather than we can find entire code fragments.

Figure 11 shows the number of concretized changes and patches from 395 Defects4j patches. Four groups of bar plots represent 10 and 100 trials with internal names, and 10

**Table 7** Number of changes and patches which can be concretized by concrete names from buggy code

	Internal	Internal + External	Total
Changes	418 (38.78%)	551 (51.11%)	1,078
Patches	116 (31.78%)	174 (47.67%)	365



**Fig. 11** Completely concretized changes and patches from defects4j

and 100 trials with both internal and external names from left to right. Since all evaluated name selection methods contain non-deterministic feature (either random or probabilistic), we tried to concretize each change 10 and 100 times repeatedly, and observed whether one of these attempts completely concretized the change. APR techniques often employs a generate-and-validate approach which generates many patch candidates, hence evaluation with multiple attempts can roughly reflect such approach.

Overall, TC, VFL and TC-VFL methods found correct names for several times more changes and patches than the baseline RND method in all cases. Among them, TC-VFL method shows the best performance. With 10 trials, TC-VFL method found correct names for 220 changes and 57 patches using internal and external names. It is 39.93% (220/551) of the changes and 32.76% (57/174) of the patches which can be concretized by the collected names. These numbers are significantly higher than RND method, which correctly found names for only 33 (5.99%) changes and 7 (4.02%) patches. TC and VFL methods also concretized 48 (27.59%) and 30 (17.24%) patches respectively, which are several times higher than RND method.

We found that adding external names are much more beneficial to TC and TC-VFL methods than the other two methods. Note that almost 60 more patches can be concretized when using external names (Table 7). Consistent to such result, TC and TC-VFL methods concretized 20 and 36 more patches with 100 trials, when external names were added as candidates. However, RND and VFL methods concretized 6 and 1 fewer patches with the same setting. The difference between these two groups of methods is consideration of type-compatibility. By checking type-compatibility, TC and TC-VFL methods can reduce candidate concrete names for selection. RND and VFL methods consider all candidate names with different selection method. These results indicate that simply providing more concrete names without proper guidance to reduce candidates is not desirable, similar to change selection results in Section 4.2. Also, promising performance of TC and TC-VFL methods show that considering type-compatibility is an effective strategy to narrow down candidates for change concretization.

- *At least 31% of the patches can be fully concretized with existing concrete names, which is higher than 11% of fully graftable commits with existing code fragments revealed in a previous study.*
- *Considering type-compatibility is effective to reduce candidate concrete names for change concretization.*

## 5 Threats To Validity

The non-deterministic nature of some methods used in experiments raises internal validity threats. First of all, some of the change selection methods and all of the name selection methods employ random or stochastic features during selection. Hence, we might obtain different results if we conducted experiments again with different seed values. However, we repeated individual trials multiple times and aggregated results to mitigate this issue.

Another issue is that we randomly selected patches for the test set and different size change pools. If our random selection provided different sets of patches, our experiment results would be different. However, this study's major findings will be more likely unchanged even if we obtained different numbers. In particular, using contexts provided significantly improved performance compared to no context in change selection, hence small variance in numbers will not change the conclusion that using contexts is beneficial. We also provide results of statistical tests to prove that the improvement of CCA is not just a coincidence, but it is statistically significant.

External validity threats of this study mostly reside in our subject selection. We collected 4,995 patches from 11 Java software projects and used them for most of the experiments. Although our subjects include software projects of various functionalities, sizes, and history, they are all open source Java projects from Apache Software Foundation. Hence collecting changes from different source of human-written patches (e.g., commercial software) may provide different results. However, our subjects have been developed and maintained by many contributors for several years, which means that they can provide abundant source for changes and they are worth to investigate.

## 6 Related Work

### 6.1 Studies on Human-written Patches and Repair models

There have been several studies which evaluated the potential and effectiveness of the idea that collecting changes from previous patches to use them for new patch generation.

Martinez et al. proposed a repair model based on repair actions collected from human-written patches (Martinez and Monperrus 2015). They used two change models to represent repair actions with change type only (CT) or change type and changed entity type (CTET), and verified that change selection based on their natural frequency distribution can be effective to compose new patches. In this study, we used much more precise change representation which increases the number of unique changes to several hundred times more than the previous study which greatly expand patch search space. We showed that using context with most-frequent first selection can also be effective even with a lot of collected changes.

Nguyen et al. investigated change repetitiveness and tested change recommendation for bug-fixes (Nguyen et al. 2013). They represented changes with old and new ASTs, which included unchanged parts, hence change's context was implicitly considered in the representation. We explicitly considered syntactic contexts and evaluated various types of contexts in CCA repair model, and we evaluated those context types not just for change selection, but also for fix location filtering and prioritization, which are key tasks of automatic program repair.

Some studies discussed about search space and its navigation issue (Long and Rinard 2016a; Qi et al. 2015; Smith et al. 2015; Le et al. 2018). In this study, we also discussed about the navigation of search space described by CCA repair model. Our study result

showed that using contexts provides effective navigation in two key dimensions of search space: changes and fix locations.

There are other studies about recurrence of code fragments and changes which imply the potential of ideas using previous patches for new patch generation (Barr et al. 2014; Gabel and Su 2010; Ray et al. 2014; Martinez et al. 2014; Zhong and Su 2015; Zhong and Meng 2018). Our study also relies on such repetitiveness and we further tested that collecting more changes can provide more new patches if it is exploited properly with contexts.

## 6.2 Automatic Program Repair

Recent APR techniques leveraging code fragments or changes obtained from past patches for patch generation.

Genesis (Long et al. 2017) infers a set of fine-grained code transforms from existing patches. It uses integer linear programming to obtain a small enough set of code transforms which can generate all training set patches. Hence this strategy carefully expands search space, so that difficulty of navigation is not increased significantly. Eventually, it tries all collected code transforms during patch generation. Unlike Genesis, CCA repair model collects all changes from past patches to expand search space regardless of navigation difficulty. However, at the time of patch generation, it only considers changes with the same context of a fix location, to reduce search areas which need to be explored in the expanded search space.

ssFix (Xin and Reiss 2017) uses code search to supply plentiful code fragments for patch generation. It identifies code fragments syntactically related to buggy code among many available code fragments. We also tested the idea of supplying necessary code fragments by changes which preserve changed code structures and using concrete names from buggy code to generate patch candidates related to buggy code.

CapGen (Wen et al. 2018) and SimFix (Jiang et al. 2018) both collect high-level abstract changes such as `insert assignment`. Since they use simple change representation, collected changes themselves do not increase navigation difficulty significantly. Collected changes are concretized using code fragments (e.g., `insert i=i+1`) from buggy code, to generate patch candidates more closely related to fix given bugs. SimFix concretizes changes with existing code fragments obtained from buggy code, by considering similarities to a target fix location. Due to the difference in change representation, CCA repair model concretizes changes using concrete names from buggy code, and our results show that finding appropriate names could be more effective strategy than finding entire code fragments. CapGen considers genealogy context, which resembles parent group contexts in this study. We observed that change selection performance is better with type or hash group contexts, hence using these context types could be helpful to CapGen for more effective patch generation.

There are techniques using semantic-based repair (DeMarco et al. 2014; Nguyen et al. 2013; Mechtaev et al. 2015, 2016; Ke et al. 2015; Le et al. 2017a, b) which often exploit SMT constraints solving to generate bug fixes. CCA repair model can complement such techniques when they synthesize code-level patches. Since CCA collects fine-grained abstract changes and concretizes them for given buggy code, it can be adapted for patch synthesis of semantic-based repair techniques.

There are other APR techniques generate patch candidates using various method and resources. PAR (Kim et al. 2013), SPR (Long and Rinard 2015), Prophet (Long and Rinard 2016b) and Relifix (Tan and Roychoudhury 2015) use pre-defined fix templates or schemas obtained from intuition or manual inspection on human-written patches. Some techniques

leverage existing code fragments to generate new patches (Le Goues et al. 2012; Weimer et al. 2013; Qi et al. 2013, 2014; Petke et al. 2014; Xin and Reiss 2017). Many other automatic program repair techniques using generated-and-validate approach also have been proposed (Debroy and Wong 2010; Weimer et al. 2009; Perkins et al. 2009; Qi et al. 2015; Goues et al. 2012; Arcuri and Yao 2008). Even for these techniques, CCA's fix location filtering and prioritization can be helpful to improve performance.

### 6.3 Other Related Work

There is another line of work related to this study, which is about mining edits and code transfer (Meng et al. 2011a, b, 2013; Barr et al. 2015; Rolim et al. 2017; Sidiroglou-Douskos et al. 2015) or identifying bug-fix patterns (Livshits and Zimmermann 2005; Martinez et al. 2013) from source code or code changes, which plays a similar role as CCA, but for different purposes. They all proposed and tested ideas of collecting changes and patterns which might be adapted for program repair like CCA.

One of the important parts of CCA is change collection which identifies AST subtree changes from software history, hence AST differencing techniques are also related to this study. Although we used a specific tool for experiments, CCA is not bound to one specific technique. Other techniques (Falleri et al. 2014; Fluri et al. 2007; Raghavan et al. 2004) can be used in CCA, as long as a technique provides AST-level changes and AST contexts can be obtained from them.

## 7 Conclusion

In this study, we evaluated the effectiveness of CCA on change selection, fix location selection and change concretization. Our findings indicate that using contexts to select necessary changes and desirable fix locations is more effective than doing that without contexts. By considering contexts, we can find more necessary changes and examine fewer fix location candidates than no context, and this advantage is maintained regardless of selection methods or context types. CCA's fine-grained changes convert change concretization into name selection problem. We also found that we can more easily find concrete names required for patches from buggy code, compared to finding the entire code fragments from buggy code.

We hope our findings in this study will be helpful to leverage contexts of changes for APR techniques. Clearly, it is beneficial to use contexts to navigate a vast, sparse search space for patch generation. We assessed CCA on each aspect of search space navigation independently and showed that using contexts is effective in all aspects. Hence using contexts only for a certain aspect still has a good possibility to enhance existing APR techniques.

## References

- Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: IEEE congress on evolutionary computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence), pp 162–168. <https://doi.org/10.1109/CEC.2008.4630793>
- Barr ET, Brun Y, Devanbu P, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, FSE '14

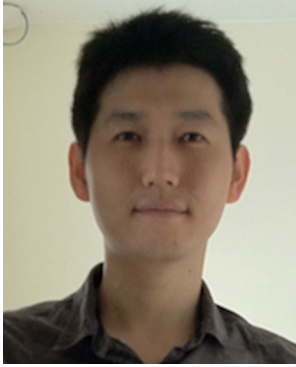


- Barr ET, Harman M, Jia Y, Marginean A, Petke J (2015) Automated software transplantation. In: Proceedings of the 2015 international symposium on software testing and analysis, ISSTA 2015. ACM, New York, pp 257–269. <https://doi.org/10.1145/2771783.2771796>
- Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E (2002) Pinpoint: Problem determination in large, dynamic internet services. In: International conference on dependable systems and networks, 2002. DSN 2002. Proceedings, pp 595–604. IEEE
- Chilowicz M, Duris E, Roussel G, Paris-est U (2009) Syntax tree fingerprinting: a foundation for source code similarity detection
- Debroy V, Wong WE (2010) Using mutation to automatically suggest fixes for faulty programs. In: Proceedings of the 2010 3rd international conference on software testing, verification and validation, ICST '10. IEEE Computer Society, Washington, pp 65–74. <https://doi.org/10.1109/ICST.2010.66>
- DeMarco F, Xuan J, Le Berre D, Monperrus M (2014) 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
- Falleri JR, Morandat F, Blanc X, Martinez M, Monperrus M (2014) Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering, ASE '14. ACM, New York, pp 313–324. <https://doi.org/10.1145/2642937.2642982>
- Fluri B, Wursch M, Pinzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans Softw Eng* 33(11):725–743. <https://doi.org/10.1109/TSE.2007.70731>
- Gabel M, Su Z (2010) A study of the uniqueness of source code. In: Proceedings of the Eighteenth ACM SIGSOFT international symposium on foundations of software engineering, FSE '10. ACM, New York, pp 147–156. <https://doi.org/10.1145/1882291.1882315>
- Goues CL, Nguyen T, Forrest S, Weimer W (2012) Genprog: A generic method for automatic software repair. *IEEE Trans Softw Eng* 38(1):54–72. <https://doi.org/10.1109/TSE.2011.104>
- Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) 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. ACM, New York, pp 298–309. <https://doi.org/10.1145/3213846.3213871>
- Just R, Jalali D, Ernst MD (2014) 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. ACM, New York, pp 437–440. <https://doi.org/10.1145/2610384.2628055>
- Ke Y, Stolee KT, Goues CL, Brun Y (2015) Repairing programs with semantic code search (t). In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE), pp 295–306
- Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 international conference on software engineering, ICSE '13. <http://dl.acm.org/citation.cfm?id=2486788.2486893>
- Kim J, Kim J, Lee E (2018) Vf: Variable-based fault localization. *Information and Software Technology*. <http://www.sciencedirect.com/science/article/pii/S0950584918302453>
- Kim J, Kim S (2016) Location aware source code differencing for mining changes. Tech. rep., Hong Kong University of Science and Technology. <https://github.com/thwak/LAS>. [Online; accessed 05-Mar-2019]
- Le XB, Lo D, Goues CL (2016) History driven program repair. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 01, pp 213–224. <https://doi.org/10.1109/SANER.2016.76>
- Le XBD, Chu DH, Lo D, Le Goues C, Visser W (2017a) Jfix: Semantics-based repair of java programs via symbolic pathfinder. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2017. ACM, New York, pp 376–379. <https://doi.org/10.1145/3092703.3098225>
- Le XBD, Chu DH, Lo D, Le Goues C, Visser W (2017b) 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. ACM, New York, pp 593–604. <https://doi.org/10.1145/3106237.3106309>
- Le XBD, Thung F, Lo D, Goues CL (2018) Overfitting in semantics-based automated program repair. *Empir Softw Eng* 23(5):3007–3033. <https://doi.org/10.1007/s10664-017-9577-2>
- Le Goues C, Dewey-Vogt M, Forrest S, Weimer W (2012) 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. IEEE Press, Piscataway, pp 3–13. <http://dl.acm.org/citation.cfm?id=2337223.2337225>
- Lipowski A, Lipowska D (2012) Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications* 391(6):2193–2196. <https://doi.org/10.1016/j.physa.2011.12.004>. <http://www.sciencedirect.com/science/article/pii/S0378437111009010>



- Liu K, Koyuncu A, Kim D, Tegawendé F, Bissyandé T (2019) 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
- Livshits B, Zimmermann T (2005) 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. ACM, New York, pp 296–305. <https://doi.org/10.1145/1081706.1081754>
- Long F, Amidon P, Rinard M (2017) Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 727–739. <https://doi.org/10.1145/3106237.3106253>
- Long F, Rinard M (2015) Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. ACM, New York, pp 166–178. <https://doi.org/10.1145/2786805.2786811>
- Long F, Rinard M (2016a) 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. ACM, New York, pp 702–713. <https://doi.org/10.1145/2884781.2884872>
- Long F, Rinard M (2016b) Automatic patch generation by learning correct code. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL '16. ACM, New York, pp 298–312. <https://doi.org/10.1145/2837614.2837617>
- Martinez M, Duchien L, Monperrus M (2013) Automatically extracting instances of code change patterns with ast analysis. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance, ICSM '13. IEEE Computer Society, Washington, pp 388–391. <https://doi.org/10.1109/ICSM.2013.54>
- Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. *Empir Softw Eng* 20(1):176–205. <https://doi.org/10.1007/s10664-013-9282-8>
- Martinez M, Weimer W, Monperrus M (2014) 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
- Mechtaev S, Yi J, Roychoudhury A (2015) Directfix: Looking for simple program repairs. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 1, pp 448–458
- Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th international conference on software engineering, ICSE '16. ACM, New York, pp 691–701. <https://doi.org/10.1145/2884781.2884807>
- Meng N, Kim M, McKinley KS (2011a) 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. ACM, New York, pp 440–443. <https://doi.org/10.1145/2025113.2025185>
- Meng N, Kim M, McKinley KS (2011b) Systematic editing: Generating program transformations from an example. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11. ACM, New York, pp 329–342. <https://doi.org/10.1145/1993498.1993537>
- Meng N, Kim M, McKinley KS (2013) 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
- Meyer ADS, Garcia AAF, Souza APD, Souza CLD Jr (2004) Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (*zea mays* l). *Genet Mol Biol* 27(1):83–91
- Nguyen HA, Nguyen AT, Nguyen T, Nguyen T, Rajan H (2013) A study of repetitiveness of code changes in software evolution. In: 2013 IEEE/ACM 28th international conference on automated software engineering (ASE), pp 180–190
- Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: Program repair via semantic analysis. In: Proceedings of the 2013 international conference on software engineering, pp 772–781. IEEE Press
- Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: Proceedings of the 39th international conference on software engineering, ICSE '17. IEEE Press, Piscataway, pp 609–620. <https://doi.org/10.1109/ICSE.2017.62>
- Perkins JH, Kim S, Larsen S, Amarasinghe S, Bachrach J, Carbin M, Pacheco C, Sherwood F, Sidiroglou S, Sullivan G et al (2009) Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles, pp 87–102. ACM
- Petke J, Harman M, Langdon WB, Weimer W (2014) Using genetic improvement & code transplants to specialise a c++ program to a problem class. In: 17th European conference on genetic programming (EuroGP), Granada, Spain

- Qi Y, Mao X, Lei Y (2013) Efficient automated program repair through fault-recorded testing prioritization. In: Proceedings of the 2013 IEEE international conference on software maintenance, ICSM '13. IEEE Computer Society, Washington, pp 180–189, <https://doi.org/10.1109/ICSM.2013.29>
- Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th international conference on software engineering, pp 254–265. ACM
- Qi Z, Long F, Achour S, Rinard M (2015) 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. ACM, New York, pp 24–36, <https://doi.org/10.1145/2771783.2771791>
- Raghavan S, Rohana R, Leon D, Podgurski A, Augustine V (2004) 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
- Ray B, Nagappan M, Bird C, Nagappan N, Zimmermann T (2014) The uniqueness of changes: characteristics and applications. Tech. rep., Microsoft Research Technical Report
- Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B (2017) Learning syntactic program transformations from examples. In: Proceedings of the 39th international conference on software engineering, ICSE '17. IEEE Press, Piscataway, pp 404–415, <https://doi.org/10.1109/ICSE.2017.44>
- Saha RK, Lyu Y, Yoshida H, Prasad MR (2017) Elixir: Effective object oriented program repair. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017. IEEE Press, Piscataway, pp 648–659, <http://dl.acm.org/citation.cfm?id=3155562.3155643>
- Sidiroglou-Douskos S, Lahtinen E, Long F, Rinard M (2015) 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. ACM, New York, pp 43–54, <https://doi.org/10.1145/2737924.2737988>
- Smith EK, Barr ET, Le Goues C, Brun Y (2015) 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. ACM, New York, pp 532–543, <https://doi.org/10.1145/2786805.2786825>
- Tan SH, Roychoudhury A (2015) Relifix: Automated repair of software regressions. In: Proceedings of the 37th international conference on software engineering - Volume 1, ICSE '15. IEEE Press, Piscataway, pp 471–482, <http://dl.acm.org/citation.cfm?id=2818754.2818813>
- Tao Y, Dang Y, Xie T, Zhang D, Kim S (2012) How do software engineers understand code changes?: An exploratory study in industry. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, FSE '12. ACM, New York, pp 51:1–51:11, <https://doi.org/10.1145/2393596.2393656>
- Tao Y, Kim S (2015) Partitioning composite code changes to facilitate code review. In: 2015 IEEE/ACM 12th working conference on mining software repositories, pp 180–190
- Weimer W, Fry ZP, Forrest S (2013) Leveraging program equivalence for adaptive program repair: Models and first results. In: 2013 IEEE/ACM 28th international conference on automated software engineering (ASE), pp 356–366. IEEE
- Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Proceedings of the 31st international conference on software engineering, pp 364–374
- Wen M, Chen J, Wu R, Hao D, Cheung SC (2018) Context-aware patch generation for better automated program repair. In: Proceedings of the 40th international conference on software engineering, ICSE '18. ACM, New York, pp 1–11, <https://doi.org/10.1145/3180155.3180233>
- Xin Q, Reiss SP (2017) Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017. IEEE Press, Piscataway, pp 660–670, <http://dl.acm.org/citation.cfm?id=3155562.3155644>
- Zhong H, Meng N (2018) 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. ACM, New York, pp 885–885, <https://doi.org/10.1145/3180155.3182550>
- Zhong H, Su Z (2015) An empirical study on real bug fixes. In: Proceedings of the 37th international conference on software engineering - Volume 1, ICSE '15. IEEE Press, Piscataway, pp 913–923, <http://dl.acm.org/citation.cfm?id=2818754.2818864>



**Jindae Kim** is a PhD student of Computer Science at the Hong Kong University of Science and Technology. He received the B.S.(2009), M.S.(2011) in computer science and engineering from Seoul National Univ., Korea. His research focuses on automatic program repair and change mining.



**Jeongho Kim** received his Ph.D. and M.S. degrees in Computer Engineering from Sungkyunkwan University, Korea, in 2019, and a B.S. degree in Industrial and Management Engineering from Hansung University, Korea, in 2012. He is currently a research scientist in the Institute of Convergence Technology of KT Corporation, Korea. His research interests include software testing, debugging, fault localization, automated program repair.




**Eunseok Lee** received his Ph.D. and M.S. degrees in Information Engineering from Tohoku University, Japan, in 1992 and 1988, respectively, and a B.S. degree in Electronic Engineering from Sungkyunkwan University, Korea, in 1985. He is a professor in the department of Computer Engineering at Sungkyunkwan University. From 1994 to 1995, he was an assistant professor in the department of Information Engineering of Tohoku University, Japan. He was a research scientist in the Information and Electronics Laboratory of Mitsubishi Electric Corporation, Japan, from 1992 to 1994. His research topics include methodologies in software engineering, autonomic computing, and web-based agent technologies.



**Sunghun Kim** is an Associate Professor of Computer Science at the Hong Kong University of Science and Technology. He got his BS in Electrical Engineering at Daegu University, Korea in 1996. He completed his Ph.D. in the Computer Science Department at the University of California, Santa Cruz in 2006. He was a postdoctoral associate at Massachusetts Institute of Technology and a member of the Program Analysis Group. He was a Chief Technical Officer (CTO), and led a 25-person team at the Nara Vision Co. Ltd, a leading Internet software company in Korea for six years.

## Affiliations

Jindae Kim<sup>1</sup>  · Jeongho Kim<sup>2</sup> · Eunseok Lee<sup>2</sup> · Sunghun Kim<sup>1</sup>

Jeongho Kim  
jeonghodot@skku.edu

Eunseok Lee  
leees@skku.edu

Sunghun Kim  
hunkim@cse.ust.hk

<sup>1</sup> The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

<sup>2</sup> Sungkyunkwan University, Seoul, Republic of Korea