Shaping Program Repair Space with Existing Patches and Similar Code*

Jiajun Jiang
Key Laboratory of High Confidence
Software Technologies, MoE
Peking University
Beijing, China
jiajun.jiang@pku.edu.cn

Yingfei Xiong
Key Laboratory of High Confidence
Software Technologies, MoE
Peking University
Beijing, China
xiongyf@pku.edu.cn

Hongyu Zhang
School of Electrical Engineering and
Computing
The University of Newcastle
Callaghan NSW, Australia
hongyu.zhang@newcastle.edu.au

Qing Gao
National Engineering Research
Center for Software Engineering
Peking University
Beijing, China
gaoqing@pku.edu.cn

Xiangqun Chen
Key Laboratory of High Confidence
Software Technologies, MoE
Peking University
Beijing, China
cherry@sei.pku.edu.cn

ABSTRACT

Automated program repair (APR) has great potential to reduce bug-fixing effort and many approaches have been proposed in recent years. APRs are often treated as a search problem where the search space consists of all the possible patches and the goal is to identify the correct patch in the space. Many techniques take a data-driven approach and analyze data sources such as existing patches and similar source code to help identify the correct patch. However, while existing patches and similar code provide complementary information, existing techniques analyze only a single source and cannot be easily extended to analyze both.

In this paper, we propose a novel automatic program repair approach that utilizes both existing patches and similar code. Our approach mines an abstract search space from existing patches and obtains a concrete search space by differencing with similar code snippets. Then we search within the intersection of the two search spaces. We have implemented our approach as a tool called *SimFix*, and evaluated it on the Defects4J benchmark. Our tool successfully fixed 34 bugs. To our best knowledge, this is the largest number of bugs fixed by a single technology on the Defects4J benchmark. Furthermore, as far as we know, 13 bugs fixed by our approach have never been fixed by the current approaches.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**;

KEYWORDS

Automated program repair, code differencing, code adaptation

1 INTRODUCTION

Automated program repair (APR) aims at reducing bug-fixing effort by automatically generating patches that satisfy a specification, which is a step forward to software automation [42]. In recent years, many automated program repair techniques have been proposed [12, 15, 23, 33, 37, 41, 46, 60, 61, 66, 69]. A typical APR approach takes a faulty program and a set of test cases as input (where the program fails at least one test case), and produces a patch to repair the fault (where the patched program at least passes all the tests).

APR is often treated as a search program where the space consists of all possible patches and the goal is to identify the correct patch from the space. The problem is challenging because the space is usually huge, and contains a lot of plausible (patches that pass all the tests) but incorrect patches [36]. Test cases cannot distinguish between correct and plausible but incorrect patches. An APR approach not only needs to locate the correct patch from a huge space, but also needs to avoid the plausible but incorrect patches.

To solve this problem, many APR techniques take a data-driven approach to estimate the likelihood of patches and confine/prioritize the search space so that the patches that are more likely to be correct are tried first. One of the commonly-used data sources is existing patches. Analyzing the existing patches gives us the (potentially conditional) distribution of bug-fixing modifications, thus

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen

we can pick the patches that make the most likely modifications. Typical approaches include PAR [26], Genesis [35], QAFix [13] and SOFix [34], which confine the search space based on repair patterns, and Prophet [37] and HDRepair [29] that rank the patches by learned models. Another commonly-used data source is source code. Analyzing the source code helps us understand the internal structures of the program to be repaired (including the common and context-specific code), thus we can select the patches that fit the local programming context. Typical approaches include GenProg [31] and its variants [18, 72], which combine statements from (similar) code snippets in the same project, as well as SearchRepair [25] that performs semantic search in a code base and ACS [68] that ranks patch ingredients based on statistical results from GitHub.

However, both data sources have their limitations. On the one hand, while existing patches provide comprehensive characterization of the bug-fixing modifications, they are often not abundant enough to cover different situations. In particular, the bug-fixing modifications for local elements in the current projects, for example, adding boundary check for a user-defined method or removing redundant initialization of a user-defined object, often cannot be found in existing patches as there usually exist only a limited number of patches from the history of current project. On the other hand, the source code does not characterize the likelihood of a bug-fixing modification, i.e., how likely the developer can make the mistake. For example, it is usually more likely to change ">" into ">=" than to insert a loop statement, but the source code alone cannot tell us this information.

In this paper, we propose to combine the two data sources to better guide automated program repair. Our basic idea is to characterize a search space using each source, and take the intersection of the two spaces. In more detail, our approach contains two stages: mining and repairing. The two stages share a search space definition S that contains all possible patches. In the mining stage, given a corpus of existing patches, the system produces a confined space $S_1 \subseteq S$ by analyzing the corpus. The patches can come from different projects, and once mined, the space S_1 can be used to repair defects in different projects. In the repairing stage, given a faulty code snippet, the system identifies the similar code snippets in the same project and produces another confined space $S_2 \subseteq S$. Then the system takes the intersection of the two spaces to get the final search space $S_1 \cap S_2$. Within the final space, the system searches patches using basic heuristics such as syntactic distance [40, 41].

To realize this approach, we need methods to produce a search space from patches and from source code, respectively. To enable cross-project mining, we define an abstract search space on AST node types, where each abstract patch in the abstract search space is an abstraction of concrete patches that modify the AST node types. Then the space is defined by the frequent abstract patches. This abstract space definition abstracts away project-specific details and thus is likely to generalize from a small set of patches. Furthermore, this space can be mined automatically and efficiently.

To obtain a search space from source code, we follow existing approaches that reuse repair ingredients from similar code [18, 25, 31, 72]: first locate the code snippets that are similar to the faulty snippet, and then combine the ingredients in the similar code snippets to form candidate patches. However, it is hard to find a proper granularity for the ingredients. If we use a coarse-grained granularity, such as statements, many bugs cannot be repaired. If we use a fine-grained granularity, such as AST nodes, their combinations may form a large space that cannot be explored efficiently. To overcome this problem, we propose a novel differencing-based approach to reduce the search space for fine-grained granularities. More concretely, we first define a set of features to calculate the syntactic distance between the faulty code snippet and other code snippets. For a similar code snippet that has a shorter distance (called a *donor*), we compare the two code snippets and obtain modifications from the faulty snippet to the donor snippet. Finally, the actual search space is formed by the combinations of the modifications. Since variable names at different snippets are often different, we also build a mapping between variable names and take the mapping into consideration when extracting and applying the modifications.

We have implemented our approach as an automated program repair tool called *SimFix*, and evaluated it on the Defects4J benchmark [22]. Our tool successfully fixed 34 defects. To our best knowledge, this is the largest number of bugs fixed by a single technology on the Defects4J benchmark. Furthermore, 13 defects that fixed by our approach have never been correctly fixed by the related techniques as far as we know.

In summary, this paper makes the following contributions:

- An automated program repair approach based on the intersection of two search spaces: the search space from existing patches and the search space from similar code.
- A method to obtain a search space from existing patches, based on an abstract space definition on AST types.
- A method to obtain a search space from similar code based on code differencing.
- An experiment on Defects4J that shows the effectiveness of our approach.

The remainder of the paper is organized as follows. Section 2 motivates our approach by examples. Section 3 illustrates our methodology in detail. Section 4 evaluates the effectiveness of our approach on Defects4J, while Section 5 and 6 discuss the limitations and related work, respectively. Finally, Section 7 concludes the paper.

2 MOTIVATING EXAMPLE

This section motivates our approach using a real-world example from our experiment. Listing 1 shows a defect, *Closure-57*, in Defects4J benchmark [22]. Note that in this paper, a line of code starting with "+" denotes a newly added line and lines starting with "-" denote lines to be deleted.

```
1 +if(target != null && target.getType()==Token.STRING){
2 -if(target != null){
3    className = target.getString();
4 }
```

Listing 1: The faulty code snippet from *Closure-57*

```
1 if(last != null && last.getType() == Token.STRING){
2    String propName = last.getString();
3    return (propName.equals(methodName));
4 }
```

Listing 2: A similar code snippet to the faulty one

In this example, the condition in the `if` statement is not complete. Before accessing the `getString()` method of the `target` object, we need to check the type of the object and only when it is `Token.STRING` could we assign its `String` value to the variable `className`. In a typical program repair process, the system first uses a fault localization algorithm to locate this snippet, and then finds a set of modifications on the snippet (i.e., a patch) in the search space that can make all tests pass.

If we allow all possible modifications, the search space is infinite and it is not easy to quickly identify the correct patch within the space. To enable efficient search, we need to shrink the search space such that only the most probable patches are included in the space.

Let us first consider how to use similar code to shrink the space. Based on existing studies [18, 25, 72], code snippets that are similar to the faulty snippet is more likely to contain ingredients for the patch. Therefore, we can confine the search space to contain only ingredients from similar code snippets. In our example, given a faulty code snippet, we find a donor code snippet (Listing 2) that is similar to the faulty code shown in Listing 1. They have similar code structures in terms of abstract syntax tree - both have an `IfStatement` surrounding a same `MethodInvocation` (`getString()`).

As mentioned in the Introduction, if we directly combine the ingredients in the donor snippet, it is difficult to find a proper granularity. For example, several existing studies [18, 25, 31, 72] use the statement granularity, i.e., inserting or replacing with statements in the donor snippet. In this example, the whole snippet is an *if* statement, and directly replacing the faulty statement with the donor statement would not repair the bug as the donor snippet uses different variable names and contains a different body. On the other hand, if we use a fine-grained level, such as AST node level, there are many possible ways to combine the AST nodes and the search space is still large.

To deal with this problem, we combine variable matching and code differencing to generate the search space. First, we introduce an algorithm that maps the variables in the two snippets based on their usages, types and names. Based on the mapping, we replace the variables in the donor code snippet with the mapped variables in the faulty snippet. In this example, `last` would be replaced by `target` because they share the same type and are both used in the `!=` comparison and in the method invocation of `getString()`. Similarly, `propName` would be replaced by `className`.

Second, we utilize a fine-grained code differencing algorithm to compare the two snippets and extract modifications from the faulty snippet to the donor snippet. Our algorithm is adapted from GumTree [10] and can extract modifications at the level of AST subtree. In the example, our algorithm extracts the following two modifications.

- Modification 1: replace the condition "`target!=null`" with "`target!=null && target.getType()==Token.STRING`".
- Modification 2: insert the statement "`return (className.equals (methodName))`" at the end of the `if` body.

Then the search space contains the combinations of all modifications. In this example, the search space includes three patches: Modification 1, Modification 2, and both of the two modifications.

Though using similar code we greatly reduce the search space, the space may still contain many plausible but incorrect patches.

For instance, the example in Listing 3 comes from the *Time-24* bug in Defects4J benchmark, which was incorrectly repaired in a comparative experiment in our evaluation (Section 4). In this example, the fault localization algorithm mistakenly recognizes a correct snippet as faulty. From this snippet we can find a donor snippet (lines 2-4) which leads to a patch that replaces a method call with a cast expression (lines 7-8), and this patch happens to pass all the tests. However, in fact it is quite rare that a developer would mix up a cast expression with a method call, and thus this patch is uncommon and less likely to be correct. In other words, if we derive a search space from existing patches to cover only common repair patterns, we could exclude this patch from the space.

```
1  // donor code
2  if(instant>=0){
3      return (int)(instant%DateTimeConst.MILLI_PER_DAY);
4  }
5  // incorrect patch on a correct snippet
6  if(instant<firstWeekMillis1){
7-     return getWeeksInYear(year-1);
8+     return (int)(instant%DateTimeConst.MILLI_PER_DAY);
9  }
```

Listing 3: A pausible but incorrect repair of *Time-24*

To obtain a space from a corpus of patches, we define abstract modifications that include only AST node types. For example, one of the abstract modifications could be "replacing a method call with a cast expression", which contains the patch mentioned above. Then we count the frequencies of the abstract modifications in the corpus, and define the search space as the combinations of the abstract modifications whose frequencies are higher than a threshold. In this example, the frequency of the above abstract modification is lower than the threshold, thus after we take intersection between the two search spaces, the incorrect patch would be excluded. Please note that we obtain the space from existing patches in the offline mining phase, and this space is used to repair many different bugs in the online repairing phase.

After we obtained the final search space, we sort the patches in the space according to three sorting rules (Section 3.4.5). e.g., patches with simpler modifications are sorted first. In our example (Listing 1), we first try the Modification 1 and the Modification 2, before we consider the combination of the two. Whenever a patch passes all the tests, we output the result. In this way, we will produce the correct patch for the first example (Listing 1) and will not produce the incorrect patch for the second example (Listing 3)

3 APPROACH

Figure 1 shows an overview of our approach. Our approach consists of an offline mining stage and an online repairing stage. In the mining stage (Section 3.3), a set of patches are analyzed to obtain the search space S_1. In the repairing stage (Section 3.4) consists of five phases. Given a faulty program, the first phase (Section 3.4.1) identifies an ordered list of suspicious faulty statements using standard fault localization approaches. For each faulty location, the second phase (Section 3.4.2) locates donor snippets that are similar to the faulty code in the same project. For each donor code snippet, the third phase (Section 3.4.3) maps variables between the donor snippet and the faulty snippet and replaces all variables in the donor snippet with the corresponding variables. The fourth

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen

phase (Section 3.4.4) diffs the two snippets, obtains the search space S_2, and computes $S_1 \cap S_2$. Finally, the final phase (Section 3.4.5) generates a list of patches within the intersection and validates them using the test suite. Both stages share the same search space definition (Section 3.1) and the same code differencing algorithm (Section 3.2), which we will introduce first in this section.

3.1 Search Space

In this section we give the definitions of search space and abstract search space. Before defining the search space, we need to be more formal about the Abstract Syntax Trees (ASTs). We define an AST as an order tree, i.e., the children of a node is ordered as a sequence, and denotes the set of all ASTs as *AST*. To avoid confusion, we directly use *node* to refer the location of a node in the AST tree. We use $parent(n)$ to denote the parent of the node n, $children(n)$ to denote the children sequence of node n, $index(n)$ to denote the index of n in the children list, and $root(t)$ to denote the root node of an AST tree t. We further assume that there exists a function $type(n)$ that assigns each node n with a node type. Also, we assume the node types fall into two distinct categories. The first category is tuple, which has a fixed number of children. For example, a node of type MethodInvocation must have three children: the receiver expression, the method name, and the arguments. The second category is sequence, whose children list is not fixed. For example, a Block node can have a sequence of statements as children, and the number of statements is not predetermined. We use $isTuple(n)$ or $isTuple(T)$ to denote that node n has a tuple type or type T is a tuple type. Finally, some leaf nodes have an associated value, e.g., SimpleName has the associated name. We use $value(n)$ to denote the value associated with n, and $value(n)$ is $\bot$ for the node with no associated value. Strictly, all these operations rely on an additional parameter: the AST tree itself. For example, $parent(n)$ should be $parent(n, t)$ where t is the AST tree containing the node n. We make this additional parameter as implicit for clarity.

Based on the above definitions, we define modifications over an AST t as follows.

Definition 3.1. A (concrete) *modification* over an AST t is one of the following operations:

- *Insert*(n, t', i): insert AST t' under node n at index i.
- *Replace*(n, t'): replace the subtree rooted at node n with AST t'.

Then a search space is a combination of modification operations.

Definition 3.2. Let t be an AST tree and M be a set of modifications over t. The *(concrete) search space* defined by M is a powerset 2^M.

Please note that our search space definition does not include deletions. As argued by existing studies [51, 58], deleting code often leads to incorrect patches, so we do not consider deletion in our search space. RQ-4 in Section 4 evaluates the effect of this decision.

Based on this definition, the modifications produced by a differencing algorithm naturally form a search space. However, this definition depends on an AST, and is not suitable for analyzing patches over different ASTs. To support the analysis of patches, we define abstract modifications and abstract search spaces as follows.

Definition 3.3. An *abstract modification* is one of the following operations.

- *INSERT*(T): insert an AST of root type T.
- *REPLACE*(T_1, T_2): replace a subtree of root type T_1 with another AST of root type T_2.

Definition 3.4. Let M^A be a set of abstract modifications. The *abstract search space* defined by M^A is a powerset 2^{M^A}.

It is easy to see that each concrete modification corresponds to an abstract modification, and we use function *abs* to perform this conversion.

$$
\begin{aligned}
abs(Insert(n, t, i)) &= INSERT(type(root(t))) \\
abs(Replace(n, t)) &= REPLACE(type(n), type(root(t)))
\end{aligned}
$$

Finally, after we obtain an abstract space from patches and a concrete space from similar code, we need to get the intersection of them. Let S^C be a concrete space and S^A be an abstract space, their intersection is defined as follows.

$$
S^C \cap S^A = \{m | m \in S^C \land abs(m) \in S^A\}
$$

3.2 Modification Extraction

There are two places in our approach that we need to extract modification. First, when analyzing patches, we need to get the modifications performed by each patch. Second, when analyzing similar code, we need to get the modifications from the faulty snippet to the donor snippet. We use the same differencing algorithm for both places. This algorithm takes two ASTs a and b as input, and produces a set of concrete modifications from a to b.

Our algorithm shares the same skeleton as GumTree [10] but is specifically tailored for our search space. We first match nodes between the two ASTs, and extract the modifications along the matching process. Intuitively, two nodes are matched, if (1) they have the same node type, and (2) all their ancestors are matched, or (3) the previous two conditions can be satisfied by inserting some nodes from the target AST to the source AST. Algorithm 1 shows the details of the matching process between two ASTs, which basically performs a top-down search to locate matched nodes satisfying the above conditions. The algorithm starts from *match* function with the roots of the two ASTs. If the two nodes can be matched (lines 2-3), we recursively check their children. Otherwise, we check whether the nodes can be matched by inserting some parent nodes in the referenced AST to the faulty AST (lines 4-6). After two nodes are matched, we check their children. Here we distinguish the two types of nodes. For tuple nodes, only the children at the corresponding positions are matched (lines 13-14). For sequence nodes, their children can be freely matched (lines 15-24).

After we match the nodes, deriving the modifications becomes direct. We check the following four conditions in the matched ASTs, and each generates a modification. In the following, we use $a \leftrightarrow b$ to denote that a and b are matched, where a is from the source AST and b is from the target snippet. We also use $tree(a)$ to denote the subtree rooted at a.

Condition: $a \leftrightarrow b \land type(a) = type(b) \land value(a) \neq value(b)$
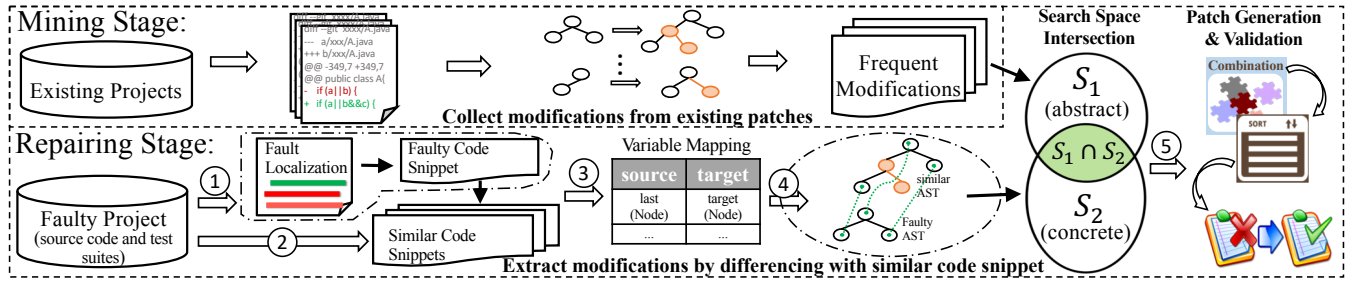 Modification: $Replace(a, tree(b))$

Figure 1: The overall workflow of the proposed approach.

Algorithm 1: Matching AST Nodes

1 **func** $match(a : ASTNode, b : ASTNode)$
2   **if** $type(a) = type(b)$ **then**
3   **return** $\{\langle a, b \rangle\} \cup matchChildren(a, b)$
4   **else**
5   **for** $each\ b' \in children(b)$ **do**
6   **if** $match(a, b') \neq \emptyset$ **then return** $match(a, b')$;
7   **return** $\emptyset$

8 **func** $matchChildren(a : ASTNode, b : ASTNode)$
9   **if** $isTuple(a)$ **then**
10   **return** $matchTuple(children(a), children(b))$
11   **else**
12   **return** $matchSet(children(a), children(b))$

13 **func** $matchTuple(as : Seq[ASTNode], bs : Seq[ASTNode])$
14   **return** $match(as.head, bs.head) \cup$
      $matchTuple(as.tail, bs.tail)$

15 **func** $matchSet(as : Seq[ASTNode], bs : Seq[ASTNode])$
16    $result \leftarrow \emptyset$
17    $toMatch \leftarrow as \times bs$
18   **while** $toMatch.size > 0$ **do**
19       $\langle a, b \rangle \leftarrow$ a pair in $toMatch$
20       $result = result \cup match(a, b)$
21   **if** $match(a, b)$ is not empty **then**
22   remove all pairs containing a or b from $toMatch$
23          $toMatch \leftarrow toMatch - \langle a, b \rangle$
24   **return** $result$

When two leaf nodes are matched but have different values, we replace the source node with the target one.

Condition: $parent(a) \leftrightarrow parent(b) \wedge isTuple(parent(a)) \wedge index(a) = index(b) \wedge a$ does not match any node

Modification: $Replace(a, tree(b))$

When two tuple nodes are matched but a pair of children at the corresponding position are not matched, we replace the source child with the target child.

Condition: $parent(a) \leftrightarrow parent(b) \wedge \neg isTuple(parent(a)) \wedge a \leftrightarrow b$

Modification: $\{Insert(parent(a), b', i) \mid b'$ is an unmatched sibling of $b \wedge i = index(b') - index(b) + index(a)\}$

When two sequence nodes are matched, we identify at least a pair of matched children, and insert all unmatched children in

the target into the source based on the relative position of the matched children.

Condition: $a \leftrightarrow b \wedge parent(a) \leftrightarrow b' \wedge b' \neq parent(b)$

Modification: $Replace(parent(a), t)$, where t is obtained by applying $Replace(b, tree(a))$ to $tree(b')$.

When a source node matches a target node in a lower level, we generate a replacement that effectively inserts the nodes above the matched target node.

3.3 Mining Stage

The input of the mining stage is a corpus of patches, where each patch includes a pair of the unpatched version and the patched version. For each patch, we use our differencing algorithm to obtain the concrete modifications, and use the *abs* function (as described in Section 3.1) to convert them into the abstract modifications. Then we count the occurrence frequency of each abstract modification. Finally, we select the abstract modifications whose occurrence frequencies are large than a threshold k to form the search space S_1. In our experiment, we set k to a value where the search space can cover 80% of the patches according to the Pareto principle [7].

3.4 Repairing Stage

3.4.1 Fault Localization. In theory, any fault localization approaches producing an ordered list of suspicious statements can be used with our approach. In our experiment on Java, we chose the Ochiai [5] algorithm implemented by Person et al. [48]. Furthermore, we use test purification [70] to preprocess the test cases to improve fault localization accuracy.

3.4.2 Donor Snippet Identification. Given a potentially faulty location, we expand it into a faulty code snippet and then locate a set of similar code snippets as donors for comparison and adaptation.

We start by defining code snippets. Given two line numbers, x and y, in a source file, a *code snippet* between $[x, y]$ is the longest sequence of statements that are included between line x and line y. Here "statements" refer to the Statement AST nodes defined in Java grammar [16]. For example, in Listing 2, the code snippet between line 1 and line 4 contains a whole if statement, while the code snippet between line 1 and line 2 includes only the statement initializing propName, as the whole if statement is not fully included in the first two lines.

To identify donor code snippets, we need to measure the similarity between two code snippets. Here we define three similarity metrics and the final similarity is the sum of the three similarities.

Structure Similarity Structure similarity concerns the structure of the ASTs. Following DECKARD [20], we extract a vector from each code snippet, where each element in the vector represents a feature of AST nodes. For example, one element could be the number of `for` statements in the code snippet, and the other element could be the number of arithmetic operators (+, -, *, /, %) in the code snippet. Then we calculate the cosine similarity [56] between the two vectors.

Variable Name Similarity Variable name similarity concerns how similar the names of the variables in the two code snippets are. We first tokenize the variable names, i.e., splitting a variable name into a set of tokens based on the CamelCase[1]. For example, the variable `className` is split into two words `class` and `name`. From the two code snippets, we can obtain two sets of tokenized variable names. Then we calculate the similarity of the two sets using Dice's coefficient[2].

Method Name Similarity Method name similarity concerns how similar the names of the methods used in the two code snippets are. Method name similarity is calculated in the same way as the variable name similarity except that we consider method names instead of variable names.

Based on the definition of code snippets, we can identify the faulty code snippet and the donor code snippets. Given a potentially faulty line, we expand it into a code snippet of size N. Assume the faulty line is n, we extract the code snippet between $[n - N/2, n + N/2 - 1]$ as the faulty snippet.

Next we identify the donor snippets. We slide a window of size N on all source files, extract the code snippet within the window, and remove the duplicate ones. In this way, we extract all possible code snippets within N lines of code. Next we calculate the similarity between faulty snippet and each donor snippet, and select the top 100 donor snippets. In our experiments we empirically set N to 10.

3.4.3 Variable Mapping. In this phase, we match variables in the faulty and donor code snippets, and build a variable mapping table for code adaptation. To do this, we leverage three kinds of similarity information, i.e., usage similarity, type similarity, and name similarity.

Usage similarity: Usage similarity captures how a variable is used within the code snippet. We first use a usage sequence to represent how a variable is used. Given a variable, we perform a post-order traversal of the AST tree, and print the parent node type for each occurrence of the variable. The result is a sequence representing the expressions that the variable has been used in. In our motivating example listed in Listing 1 and 2, the variables `target` and `last` have the following usage sequences:

> `target: [INFIX_EXP, METHOD_EXP]`
> `last: [INFIX_EXP, METHOD_EXP, METHOD_EXP]`

Then we calculate the similarity of two usage sequences based on the longest common subsequences (LCS)[3].

Type similarity: When we reuse a code fragment from the donor code snippet, we need to replace the variables with the target variables. Consequently, it is important to ensure the mapped variables have compatible types. When a variable is used as left-value

---

[1]https://en.wikipedia.org/wiki/Camel_case
[2]https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient
[3]https://en.wikipedia.org/wiki/Longest_common_subsequence_problem

in the donor code snippet, we need to ensure the type of the target variable is a super type of the original variable. When a variable is used as right-value in the donor code snippet, we need to ensure the type of the target variable is a sub type of the original variable.

Ideally, the above two conditions should be enforced when we build variable mapping. However, since at the current stage we do not know which code fragment will be reused from the code snippet, we simply use type similarity to measure the degree of compatibility of their types. Concretely, we define type similarity as a binary value. When one variable is the sub/super type of the other variable, their similarity is 1, otherwise is 0.

In Java, the primitive types do not have explicit subtyping relation. Given two primitive types T and T', we consider T is a sub type of T' if any value in type T can be losslessly converted into values in type T'.

Name similarity: The name similarity of two variables is calculated in the same way as the variable name similarity used for identifying donor code snippets, except that here we consider only two variables instead of all variables in the snippets.

The final similarity score is a weighted sum of the above three scores. After we calculate the similarities between each pair of variables, we greedily select the pairs with the highest similarities until there is no pair left. For example, the matching variables derived from Listing 1 and 2 are `last` and `target`, and `propName` and `className`. Then, we replace all variables in the donor code snippet with the corresponding mapping variables to obtain an adapted donor code snippet using shared variables with the faulty one. Later, we will refer to the donor code snippet as the adapted one without further explaining.

3.4.4 Modification Extraction and Intersection. In the forth phase, given the donor code snippets, we compare them with the faulty code snippet one by one in the descending order of similarity and extract concrete modifications using the code differencing algorithm introduced in Section 3.2. Then we identify the intersection between this set of modifications and those extracted by analyzing existing patches (Section 3.3). After which we obtain a set of concrete modifications for patch generation.

3.4.5 Patch Generation and Validation. The set of modifications obtained in the previous phase defines a reduced search space. The goal of this phase is to locate a patch in the space that passes all tests. As mentioned before, we sort the patches in the space and validate them one by one. We perform a multi-level sorting on the patches according to the following rules, in the order of their appearance in the sequence: earlier ruler indicates a higher level.

(1) Patches that contain consistent modifications are ranked higher. Consistent modifications (changes) [17, 32] are those modifications that require the same changes to a variable at different locations at the same time.
(2) Patches with fewer modifications are ranked higher.
(3) Patches with more replacements are ranked higher than patches with more insertions.

The first rule handles a special case. When a variable is replaced with the other variable, it should be replaced by the same variable at other occurrences consistently. The second and the third rules

are inspired by syntactic distance [30, 40]: the patches with fewer, simpler modifications are ranked higher.

Having obtained an ordered list of candidate patches, we validate patches one by one with the tests, and return the first patch that passes all the tests.

4 EVALUATION

To evaluate the effectiveness of our approach, we have developed a prototype tool called *SimFix*. The tool is implemented in Java and is publicly available at our project website [19].

4.1 Experiment Setup

We evaluated *SimFix* on Defects4J [22] v1.0.0[4], which is a commonly-used benchmark for automatic program repair research. Table 1 shows statistics about the projects.

Table 1: Details of the experiment benchmark.

Project	Bugs	kLoC	Tests
JFreechart (Chart)	26	96	2,205
Closure compiler (Closure)	133	90	7,927
Apache commons-math (Math)	106	85	3,602
Apache commons-lang (Lang)	65	22	2,245
Joda-Time (Time)	27	28	4,130
Total	357	321	20,109

In the table, column"**Bugs**" denotes the total number of bugs in Defects4J benchmark, column"**kLoC**" denotes the number of thousands of lines of code, and column "**Tests**" denotes the total number of test cases for each project.

Executing *SimFix* requires a set of patches such from which an abstract search space can be obtained. To obtain the set of patches, we selected three large open source projects, i.e., Ant [1], Groovy [2] and Hadoop [3]. A statistics of the three projects can be found in Table 2. Following the existing approaches [21, 47], we first developed a script to automatically analyze each commit message in the version control system and used keywords (e.g., "fix" and "repair") to select the commits that are related to bug fixing. Then we manually analyzed the remaining commit messages to further exclude unrelated commits, such as "fix doc" or "repair format", etc. In this way, 925 commit records were identified. For each commit, we extract all changed files before and after the repair.

Table 2: Projects for abstract modification extraction.

Project	kLoC	Commits	Identified Patches
Ant	138	13,731	270
Groovy	181	14,532	402
Hadoop	997	17,539	253
Total	1,316	45,802	925

In the table, column"**kLoC**" denotes the number of thousands of lines of code, column "**Commits**" denotes the number of commits in the version control system, and column "**Identified Patches**" denotes the number of patches related to bug repair.

Our experiment was conducted on a 64-bit Linux server with two Intel(R) Xeon CPUs and 128GB RAM. For each bug, we assigned two

[4]https://github.com/rjust/defects4j/releases/tag/v1.0.0

CPU cores and 8GB RAM and the repair process will terminate when a patch passes all test cases or the execution exceeds five hours. Finally, we manually examine the patches generated by *SimFix* and consider a patch correct if it is the same with or semantically equivalent to the standard patch provided by Defects4J.

4.2 Research Questions and Results

RQ 1. What are the frequent abstract modifications?

We invoked the mining stage on the set of patches to obtain an abstract search space, and then we checked the result. Table 3 shows the frequent abstract modifications defining the abstract search space. As we can see from the table, the top 3 most frequent modifications are to insert an if statement, to replace a method invocation, and to insert a method call. This result is consistent with the results in existing studies [8, 26, 39, 62]. For example, the top 5 most frequent insertions/updates identified by Martinez and Monperrus [39] are included in our space as well, such as inserting method invocations and if statements. These modifications form a small abstract space. There are in total 1640 kinds of abstract modifications formed by around 40 kinds of AST node types for Java (40 × 40 kinds of replacements and 40 kinds of insertions). However, from existing patches, there are only 16 frequent modifications shown in the table, which achieved a 102.5x reduction in the space of abstract modifications.

Table 3: Percentage of frequent operations in existing patches from open source projects listed in Table 2.

Replacement		Insertion	
(MI, MI)	21.62%	IFSTMT	22.05%
(INFIXE, INFIXE)	8.54%	ESTMT(MI)	14.70%
(NAME, NAME)	5.40%	VDSTMT	11.67%
(NAME, MI)	3.89%	ESTMT(A)	5.73%
(INFOP, INFOP)	2.05%	TRYSTMT	2.49%
(TYPE, TYPE)	1.84%	RETSTMT	1.51%
(SLIT, SLIT)	1.84%	TRSTMT	1.19%
(BLIT, BLIT)	1.18%		
(NULIT, NAME)	1.08%		

MI : MethodInvocation   IFSTMT : IfStatement
NAME : Name   ESTMT(MI) : ExpressionStatement(MethodInvocation)
INFIXE : InfixExpression   ESTMT(A) : ExpressionStatement(Assignment)
TYPE : Type   VDSTMT : VariableDeclarationStatement
SLIT : StringLiteral   INFOP : InfixExpression.Operator
NULIT : NumberLiteral   RETSTMT : ReturnStatement
BLIT : BooleanLiteral   TRYSTMT : TryStatement
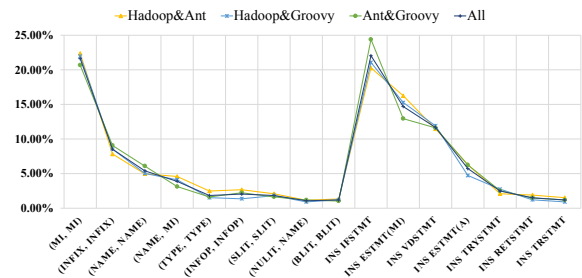TRSTMT : ThrowStatement

Figure 2: Percentages of operations for different combinations of open source projects listed in Table 2.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen

To further understand whether our set of patches is large enough to cover a representative set of changes, we analyzed the change distributions with different number of projects. Then we compared the result space with the original space to see how many abstract modifications have changed. We enumerated all the three combinations of two projects and found the abstract spaces are all identical to the original one. To further understand the differences between different patch sets, we compared the percentages of frequent abstract modifications in the patches, as shown in Figure 2. As we can see from the Figure, the percentages from different patch sets are quite close. These results indicate that our dataset is large enough. Also, our abstract space can be mined from a small set of patches, and generalizes well among projects.

RQ 2. How effective is *SimFix* on Defects4J?

This RQ evaluates the performance of *SimFix* on Defects4J. We shall use the abstract search space we obtained in RQ-1. The result is compared with eight state-of-the-art automated program repair techniques. We chose the eight techniques because they were also evaluated on Defects4J and the results can be directly compared. The header of Table 4 lists the eight comparative techniques.

The evaluation results are shown in Table 4, where each cell denotes the number of bugs fixed by the corresponding repair technique on each project. Besides, for each project, we highlighted the corresponding techniques with the most number of bugs fixed. Please note that some papers only report bugs fixed by the first patch, while some reports the bugs fixed by any patch regardless of the rank. The numbers outside the parentheses indicate the bugs fixed by the first patch while the numbers inside parentheses indicate the bugs fixed by any patch. The missing numbers are marked with "-" or directly omitted.

From the table we can see that *SimFix* successfully fixed **34** bugs with the first patch in the Defects4J benchmark, achieving the most number of correct patches against all the other comparative approaches. Besides, for four projects, *SimFix* repairs the highest numbers of bugs with the first plausible patch.

Table 4: Correct patches generated by different techniques.

Proj.	SimFix	jGP	jKali	Nopol	ACS	HDR	ssFix	ELIXIR	JAID
Chart	**4**	0	0	1	2	-(2)	3	**4**	2(4)
Closure	**6**	0	0	0	0	-(7)	2	0	5(9)
Math	**14**	5	1	1	12	-(7)	10	12	1(7)
Lang	**9**	0	0	3	3	-(6)	5	8	1(5)
Time	1	0	0	0	1	-(1)	0	**2**	0(0)
Total	**34**	5	1	5	18	13(23)	20	26	9(25)

We adopted the experimental results for jGenProg (jGP), jKali and Nopol reported by Martinez et al. [38]. The results of other approaches come from the corresponding research papers, i.e., ACS [68], HDRepair (HDR) [29], ssFix [65], ELIXIR [54] and JAID [9].

To further understand the distribution of correct and incorrect patches, we have presented the experimental results in detail for each approach in Figure 3[5]. From the figure we can see that, the portion of incorrect patches *SimFix* generated is relatively small in general. In total, *SimFix* generated 22 incorrect patches along with

---

[5]There is no HDRepair because the number of incorrect patches generated by HDRepair are not reported in the paper [29].

34 correct patches, leading to a precision of 60.7%. The precision is significantly higher than most approaches (4.5%-33.3%), is close to EXILIR(63.4%), and is noticeably lower than ACS (78.3%). Please note that ACS is a technique specifically designed for precision. In addition, several approaches [64, 67, 71] have been proposed to classify patches to increase precision, and *SimFix* can potentially be combined with these approaches to gain a better precision.

Moreover, to understand how many bugs fixed by our approach can also be fixed by the existing techniques, we have summarized the overlaps among the results of the comparative techniques in a Venn diagram, as shown in Figure 4. Here we consider the bugs fixed by any patch, not just the first one. From the figure we can see that 13 bugs repaired by *SimFix* were never reported as correctly fixed by any other techniques. Combining Figure 4 and Table 4, we can see that *SimFix* is an effective approach complementary to the existing techniques.

RQ 3. How effective is the space reduction based on existing patches?

In our approach, we first obtain a concrete space from similar code, and then we reduce it by using the abstract space from existing patches. To understand how effective this step is, we implemented a variant of *SimFix*, called *SimFix-A*. It repairs programs using all candidate modifications extracted from similar code differencing even though they are not in the abstract space of existing patches.

The experimental result is shown in Table 5. From the table we can see that without the abstract space mined from existing patches, 12 less bugs were repaired. Two major factors contributed to this reduction. (1) The space contains more plausible patches that may be generated before the correct patch. From Figure 3 we can see that more plausible but incorrect patches were generated by *SimFix-A*. We have already seen the example of *Time-24* demonstrated in Section 2. (2) With a large space, more time is needed to generate and validate the patches, and the correct patch may not be generated within the time limit. In our experiment, *SimFix-A* on average explored about 2.3x more candidate modifications than *SimFix* for those faults successfully repaired by both of them, causing *SimFix-A* spent around 2.0x as much time as *SimFix* to produce a correct patch. Therefore, the reduction of our abstract search space to the concrete search space is significant.

Table 5: Comparison among variants of *SimFix*.

Approach.	Chart	Closure	Math	Lang	Time	*Total*
SimFix	4	6	14	9	1	34
SimFix-A	2	2	11	7	0	22
SimFix-D	3	6	11	9	0	29

RQ 4. How do deletion operations affect automatic program repair?

As explained in Section 3.1 that we excluded deletions from our space. In this research question, we explore the effect of this design decision. We implemented another variant of *SimFix*, called *SimFix-D*, where we include deletions into the search space and modify the differencing algorithm such that deletions are generated. The final result is listed in Table 5 and Figure 3. *SimFix-D* repaired 5 less bugs and its precision decreases 14 percentage points.
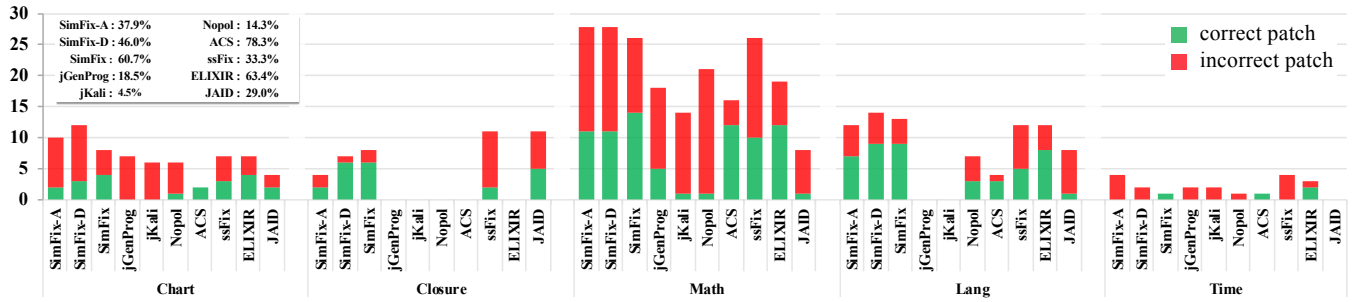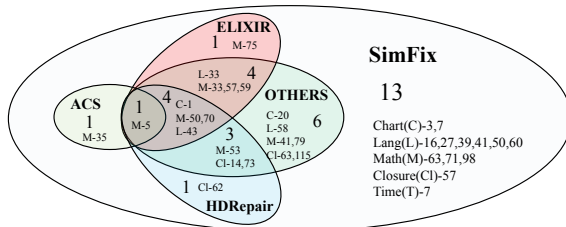
Figure 3: Statistics of the repair results for different techniques on Defects4J benchmark.

For simplicity, the diagram only shows some of the approaches listed in Table 4, and the OTHERS category includes the remaining. Besides, the Venn diagram only includes those bugs that has been successfully fixed by $SimFix$, for the whole repaired bugs for each technique, please refer to Table 4.

Figure 4: The overlaps among different techniques.

The reason of the performance loss is similar to $SimFix$-A: the enlarged search space contains more patches and more plausible (but incorrect) patches.

RQ 5. How effective is the fine-grained granularity?

Different from most existing approaches [25, 31, 49, 50, 60, 72] that reuse similar code at the statement granularity, our approach utilizes a fine-grained differencing algorithm to reuse similar code at the AST subtree granularity. To investigate how important this fine-grained granularity is, we manually analyzed the patches produced by our approach to see how many bugs can still be repaired if only permitting Statement level code reusing like existing approaches. After examining all the patches, we found that 17 less bugs would be repaired. The result suggests that the fine-grained code differencing and reusing contributed significantly to the effectiveness of our approach.

5 DISCUSSION

Size of the patches. Due to the exponential increase of the search space, program repair techniques seldom repair bugs that require large patches, i.e., patches that contains many modifications. However, when examining the patches generated by $SimFix$, we found that some large and correct patches were generated. For example, the listing below (Listing 4) shows a patch for Math-71 in Defects4J benchmark. This patch inserts four statements to fix the bug. This observation suggests that utilizing similar code with fine-grained differencing is effective in reducing the search space and has the potential to scale up patch size. Future work is needed to better understand its potential.

```
if(Math.abs(dt) <= Math.ulp(stepStart)){
+    interpolator.storeTime(stepStart);
+    System.arraycopy(y, 0, yTmp, 0, y0.length);
+    hNew = 0;
+    stepSize = 0;
     loop = false;
} else {...}
```

Listing 4: The correct patch of Math-71 generated by $SimFix$

Generalizability of the results. Our experiment was only conducted on the Defects4J benchmark, which is a widely-used dataset in automatic program repair research. Defects4J consists of five large projects developed by different developers and contains 357 bugs in total. Though in general Defects4J is a very high quality framework, most of the projects are either compilers or programming libraries, and it is not clear that the defects on these projects could represent the defects on, for example, small client applications. In particular, when the project size shrinks, it is yet unknown how many similar snippets exist for defect repair. These problems remain as future work to be explored.

Incorporating richer information. Our current implementation mainly relies on program original syntactic information without code normalization [27, 52, 59]. Especially, we treat code as ASTs and derive changes by performing tree-based differencing. This design ignores other potentially useful information, such as the dependency between variables and statements, more precise type information based on type inference, or the alias information between variables. Such richer information may be utilized to create better heuristics for code differencing and adaptation, or to filter patches earlier to avoid the heavy validation from test cases.

6 RELATED WORK

6.1 Automatic Program Repair

In recent years, automated program repair has been an active research field. Many promising approaches have been proposed and evaluated. In this section, we will compare our approach with some existing techniques that are most related to our approach. For a more complete survey of automated program repair techniques, readers are redirected to recent surveys [14, 45].

The work of ssFix [65] is in parallel with ours and is very related. The ssFix approach also uses a differencing algorithm to reuse similar code in a fine-grained granularity. Different from $SimFix$, ssFix does not utilize existing patches to reduce the search space,

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen

which, as RQ-3 shows, leaded to 12 (54.5%) more repaired bugs in our experiments. Furthermore, though both approaches apply differencing to reuse similar code, the processes are different on multiple aspects. First, ssFix searches similar code from a code base, while *SimFix* searches within the same project and does not require a code base. Second, ssFix depends on the Apache Lucene search engine [4] to collect similar code snippets, which is designed for plain text search but programs. On the contrary, our approach utilizes a feature-vector based similar code identification, and can flexibly capture different aspects. Third, ssFix only permits one modification in a patch, while *SimFix* allows the combinations of modifications. We believe these differences leaded to the performance difference between the two approaches in our evaluation (14 more repaired bugs and 27.4% increase in precision from *ssFix* to *SimFix*).

Another related work is GenProg [31, 61], which applies genetic programming to mutate existing source code for patch generation. However, it is significantly different with ours as GenProg only utilizes existing code without the guidance of history patches. Moreover, the method of reusing existing code is quite different with ours in several aspects: (1) it does not identify the similarity between different code snippets; (2) it mutates existing program elements at statement level; and (3) it does not perform adaptation when utilizing existing code snippets. As presented in RQ-5, code adaptation and fine-grained AST matching are necessary for reusing existing code snippets. RSRepair [49] changed the genetic algorithm of GenProg to random search and then Ji et al. [18] improved RSRepair by leveraging code similarity when searching inserted code snippets. Yokoyama et all. [72] adopted a similar idea that selects code lines in code regions similar to the faulty code regions for patch generation. However, all of them reuse similar code snippet at Statement level, which is coarse-grained. Besides, neither of them performs code adaptation when generating patches. On the contrary, our approach not only permits more flexible and fine-grained program element reuse but also performs code adaptation. Moreover, besides similar code, our approach combines existing patches to further reduce the search space.

SearchRepair [25] considers existing code reusing as well and it performs variable renaming with constraint solving, which is difficult to scale to complex programs. Besides, it does not utilize existing patches. Recently, White et al. [63] proposed an automatic program repair approach with deep learning, DeepRepair, which depends on a neural network model to identify similarity between code snippets. However, DeepRepair only replaces variables that are out of scope at the buggy location but not a thorough replacement and similarly, it does not incorporate external information, i.e., existing patches, to guide patch generation either. Besides, the experiment result shows that it did not find significantly more patches than the jGenProg baseline. Similarly, Prophet [37] learns a patch ranking model using machine learning algorithm based on existing patches. On the contrary, *SimFix* learns an abstract search space. Moreover, Prophet does not utilize similar code. CodePhage [57] utilizes exiting code to repair missing if-condition faults, which is different with *SimFix* that targets to general faults. Besides, CodePhage requires the donor program to accept the same input as the faulty program while *SimFix* does not. Recently, Barr et al. [6] proposed μScalpel to autotransplant code from one system into another. However, it concerns the problem of inserting the whole donor snippet while *SimFix* concerns about the changing of faulty snippet based on the donor. Also, μScalpel requires the donor snippet to be specified but *SimFix* does not. Furthermore, both CodePhage and μScalpel do not utilize within-project code like *SimFix*.

PAR [26] and Anti-pattern [58] manually define a set of patterns for patch generating or filtering based on repair history. Similarly, HDR [29] and ELIXIR [54] defined a set of repair operations or expressions based on existing bug fixes, and furthermore, they use a learnt model to prioritize candidate patches. All these approaches are different with ours since our approach automatically extracts modifications based on existing patches and similar code and then intersects these two spaces for precise candidate patch exploring.

6.2 Similar Code Analysis

Our approach depends on the search of similar code in a project, which is related to the work on code clone detection [24, 28, 32, 55]. In particular, the structure similarity used in our approach is inspired by DECKARD [20], a fast clone detection approach.

Many existing techniques dedicate to the identification of the differences between two code snippets and the generation of the transformations from one to the other. ChangeDistiller [11] is a widely-used approach for source code transformation at AST level. GumTree [10] improves ChangeDistiller by removing the assumption that leaf nodes contains a significant amount of text. In our approach we implemented a tree matching algorithm that is similar to those approaches, but nevertheless is designed for the modifications we considered.

6.3 Extracting Transformations from Patches

A number of existing approaches [13, 35, 43, 44, 53] aim to automatically extract transformations from existing patches. The goal of these approaches is to extract actionable repair patterns in these patches so that the repair patterns can be applied in different places to repair the same type of bugs. On the other hand, the main goal of our abstract space design is to exclude the unlikely patches by analyzing a small set of patches. As a result, our abstract space definition is more coarse-grained and is not actionable. Nevertheless, our abstract space can generalize from a small set of patches and plays an important role to the overall performance as shown by RQ-3 in the evaluation.

7 CONCLUSION

In this paper, we propose a novel automatic program repair approach, which is based on fine-gained code differencing and utilizes both existing patches and similar code. More concretely, by analyzing existing patches, we obtain a set of frequent abstract modifications, which form an abstract space for program repair. By analyzing similar code snippets in the same program we extract concrete modifications, which forms a concrete space. Finally, we use the intersections between the two spaces and perform fine-grained code adaptation for patch generation. We have implemented a prototype of our approach, called *SimFix*, and evaluated it on Defects4J. *SimFix* successfully fixed 34 bugs in total, where 13 has never been fixed by existing techniques. *SimFix* are publicly available at [19].

REFERENCES

[1] 2017. Apache Ant. https://github.com/apache/ant. (2017).
[2] 2017. Apache Groovy. https://github.com/apache/groovy. (2017).
[3] 2017. Apache Hadoop. https://github.com/apache/hadoop. (2017).
[4] 2017. Apache Lucene. https://lucene.apache.org. (2017).
[5] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization (PRDC). IEEE Computer Society, Washington, DC, USA, 39–46. https://doi.org/10.1109/PRDC.2006.18
[6] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation (ISSTA). ACM, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796
[7] George EP Box and R Daniel Meyer. 1986. An analysis for unreplicated fractional factorials. Technometrics 28, 1 (1986), 11–18.
[8] E. C. Campos and M. d. A. Maia. 2017. Common Bug-Fix Patterns: A Large-Scale Observational Study. In 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). 404–413. https://doi.org/10.1109/ESEM.2017.55
[9] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In ASE. https://doi.org/10.1109/ASE.2017.8115674
[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In ASE. 313–324. https://doi.org/10.1145/2642937.2642982
[11] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering (Nov 2007), 725–743.
[12] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In ICSE.
[13] Qing Gao, Hansheng Zhang, Jie Wang, and Yingfei Xiong. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In ASE. 307–318.
[14] Daniela Micucci Gazzola, Luca and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. TSE PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2755013
[15] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT (TACAS'11/ETAPS'11). 173–188.
[16] James Gosling. 2000. The Java language specification. Addison-Wesley Professional.
[17] Patricia Jablonski and Daqing Hou. 2007. CReN: A Tool for Tracking Copy-and-paste Code Clones and Renaming Identifiers Consistently in the IDE. In Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '07). ACM, New York, NY, USA, 16–20. https://doi.org/10.1145/1328279.1328283
[18] T. Ji, L. Chen, X. Mao, and X. Yi. 2016. Automated Program Repair by Using Similar Code Containing Fix Ingredients. In COMPSAC, Vol. 1. 197–202.
[19] Jiajun Jiang. 2017. SimFix. https://github.com/xgdsmileboy/SimFix. (2017).
[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones (ICSE '07). 96–105.
[21] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In PLDI. ACM. https://doi.org/10.1145/2254064.2254075
[22] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In ISSTA. 437–440.
[23] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. MintHint: Automated Synthesis of Repair Hints. In ICSE. 266–276. https://doi.org/10.1145/2568225.2568258
[24] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28 (Jul 2002), 654–670.
[25] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. 2015. Repairing Programs with Semantic Code Search (T). In ASE. 295–306. https://doi.org/10.1109/ASE.2015.60
[26] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In ICSE. 802–811.
[27] Raghavan Komondoor and Susan Horwitz. 2000. Semantics-preserving procedure extraction. In POPL. ACM, 155–169.
[28] R. Koschke, R. Falke, and P. Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In 2006 13th Working Conference on Reverse Engineering. 253–262.
[29] Xuan-Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In SANER. 213–224. https://doi.org/10.1109/SANER.2016.76
[30] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In ESEC/FSE. 593–604. https://doi.org/10.1145/3106237.3106309
[31] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. TSE 38, 1 (Jan 2012), 54–72.
[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: finding copy-paste and related bugs in large-scale software code. TSE 32 (March 2006), 176–192.
[33] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In ICST. https://doi.org/10.1109/ICST.

2013.24
[34] Xuliang Liu and Hao Zhong. 2018. Mining StackOverflow for Program Repair. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. 118–129. https://doi.org/10.1109/SANER.2018.8330202
[35] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In ESEC/FSE. 727–739. https://doi.org/10.1145/3106237.3106253
[36] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In ICSE. 702–713. https://doi.org/10.1145/2884781.2884872
[37] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 298–312. https://doi.org/10.1145/2837614.2837617
[38] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. Empirical Software Engineering 22, 4 (01 Aug 2017), 1936–1964.
[39] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. Empirical Softw. Engg. (2015), 176–205. https://doi.org/10.1007/s10664-013-9282-8
[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In ICSE. 448–458. https://doi.org/10.1109/ICSE.2015.63
[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In ICSE.
[42] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? Science China Information Sciences 61(5), 056101 (2018). https://doi.org/10.1007/s11432-017-9355-3
[43] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Sydit: Creating and Applying a Program Transformation from an Example (ESEC/FSE '11). 440–443. https://doi.org/10.1145/2025113.2025185
[44] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples (ICSE '13). 502–511.
[45] Martin Monperrus. 2017. Automatic Software Repair: a Bibliography. Technical Report. 1–24 pages. https://doi.org/10.1145/3105906
[46] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-oriented Programs (ICSE). 315–324.
[47] Thanaporn Ongkosit and Shingo Takada. 2014. Responsiveness Analysis Tool for Android Application. In Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile (DeMobile 2014). ACM. https://doi.org/10.1145/2661694.2661695
[48] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization (ICSE '17). 609–620.
[49] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In ICSE. 254–265. https://doi.org/10.1145/2568225.2568254
[50] Yuhua Qi, Xiaoguang Mao, Yanjun Wen, Ziying Dai, and Bin Gu. 2012. More efficient automatic repair of large-scale programs using weak recompilation. SCIENCE CHINA Information Sciences 55, 12 (2012), 2785–2799.
[51] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In ISSTA. 257–269.
[52] Donald B Roberts. 1999. Practical Analysis for Refactoring. Technical Report. Champaign, IL, USA.
[53] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In ICSE. 404–415.
[54] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In ASE. IEEE Press. http://dl.acm.org/citation.cfm?id=3155562.3155643
[55] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In ICSE. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877
[56] Gerald Salton (Ed.). 1988. Automatic Text Processing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[57] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic Error Elimination by Horizontal Code Transfer Across Multiple Applications (PLDI '15). 43–54.
[58] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In FSE. https://doi.org/10.1145/2950290.2950295
[59] Chenglong Wang, Jiajun Jiang, Jun Li, Yingfei Xiong, Xiangyu Luo, Lu Zhang, and Zhenjiang Hu. 2016. Transforming Programs between APIs with Many-to-Many Mappings. In ECOOP. 25:1–25:26.

[60] W. Weimer, Z.P. Fry, and S. Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE*. 356–366. https://doi.org/10.1109/ASE.2013.6693094

[61] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[62] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE*.

[63] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk. 2017. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. *ArXiv e-prints* (July 2017). arXiv:cs.SE/1707.04742

[64] Qi Xin and Steven Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA*. 226–236. https://doi.org/10.1145/3092703.3092718

[65] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-related Code for Automated Program Repair *(ASE)*. http://dl.acm.org/citation.cfm?id=3155562.3155644

[66] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. 2009. Supporting automatic model inconsistency fixing. In *ESEC/FSE*. 315–324.

[67] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *ICSE*.

[68] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *ICSE*. https://doi.org/10.1109/ICSE.2017.45

[69] Yingfei Xiong, Hansheng Zhang, Arnaud Hubaux, Steven She, Jie Wang, and Krzysztof Czarnecki. 2015. Range fixes: Interactive error resolution for software configuration. *Software Engineering, IEEE Transactions on* 41, 6 (2015), 603–619.

[70] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *FSE*. New York, NY, USA, 52–63. https://doi.org/10.1145/2635868.2635906

[71] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better Test Cases for Better Automated Program Repair. In *FSE*. 831–841. https://doi.org/10.1145/3106237.3106274

[72] Haruki Yokoyama, Yoshiki Higo, Keisuke Hotta, Takafumi Ohta, Kozo Okano, and Shinji Kusumoto. 2016. Toward Improving Ability to Repair Bugs Automatically: A Patch Candidate Location Mechanism Using Code Similarity *(SAC '16)*. 1364–1370.