Mining StackOverflow for Program Repair

Xuliang Liu
School of Software
Shanghai Jiao Tong University, China
deongaree@sjtu.edu.cn

Hao Zhong*
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

***Abstract*—In recent years, automatic program repair has been a hot research topic in the software engineering community, and many approaches have been proposed. Although these approaches produce promising results, some researchers criticize that existing approaches are still limited in their repair capability, due to their limited repair templates. Indeed, it is quite difficult to design effective repair templates. An award-wining paper analyzes thousands of manual bug fixes, but summarizes only ten repair templates. Although more bugs are thus repaired, recent studies show such repair templates are still insufficient.**

We notice that programmers often refer to Stack Overflow, when they repair bugs. With years of accumulation, Stack Overflow has millions of posts that are potentially useful to repair many bugs. The observation motives our work towards mining repair templates from Stack Overflow. In this paper, we propose a novel approach, called SOFIX, that extracts code samples from Stack Overflow, and mines repair patterns from extracted code samples. Based on our mined repair patterns, we derived 13 repair templates. We implemented these repair templates in SOFIX, and conducted evaluations on the widely used benchmark, Defects4J. Our results show that SOFIX repaired 23 bugs, which are more than existing approaches. After comparing repaired bugs and templates, we find that SOFIX repaired more bugs, since it has more repair templates. In addition, our results also reveal the urgent need for better fault localization techniques.

***Index Terms*—program repair, Stack Overflow, repair template**

I. INTRODUCTION

Although the research on automatic program repair (*e.g.*, [42]) draws much attention from both academia and industry, recent studies [36], [26] show that existing automatic-program-repair approaches are still limited in their repair capability. Zhong and Su [57] complain that it is infeasible to repair many bugs, since existing approaches provide limited repair templates. For example, the well-known tool, GenProg [42], supports only three types of coarse-grained repair templates such as inserting statements, swapping statements, and deleting statements. To handle the limitation, Kim *et al.* [17] summarize ten additional repair templates from thousands of human-written patches. However, their repair templates are still limited, both in number and granularity. It is desirable to infer repair templates from more sources.

Liu *et al.* [21] show that it is useful to link Stack Overflow threads with reported bugs. Gao *et al.* [10] show that reusing code samples in Stack Overflow is able to repair some bugs, although they did not mine any new repair templates. Recently, Stack Overflow released its posts as online archives[1]. Motivated by the above positive results and due to the availability of new sources, in this paper, we present the first attempt to mine fine-grained repair templates from Stack Overflow. To realize our vision, we have to overcome the following challenges:

Challenge 1. It is challenging to effectively extract information from Stack Overflow, since the data set contains more than 30 million posts. It is even challenging to determine which thread is worth mining for repair templates.

Challenge 2. It is challenging to mine fine-grained repair templates. Although such templates are useful, it requires accurate analysis in mining. However, code samples in Stack Overflow are typically partial programs, and only several tools provide limited analysis support.

To handle the first challenge, we consider only threads with both buggy and correct code samples, since such threads are more informative. From such threads, our tool links buggy code samples with fixed code samples. Each pair corresponds to a set of feasible repair actions, and our tool further mines repair templates from such actions. To handle the second challenge, we build Abstract Syntax Trees (ASTs) from each pair of code samples, and mine repair patterns from sequences of AST modifications. From such patterns, we derive our repair templates. This paper makes the following contributions:

- The first approach, called SOFIX, that mines repair patterns from Stack Overflow, and leverages repair templates that are derived from mined patterns to repair new bugs. SOFIX compares code samples in questions and answers for fine-grained modifications, and mines repair patterns from such modifications.
- From 31,017,891 Stack Overflow posts, SOFIX mined 136 patterns. From these patterns, we manually derived 13 repair templates. Our templates contain repair values from Stack Overflow, and 2 of them are never reported. We implement our templates in SOFIX to repair bugs.
- Evaluations on the widely used benchmark, Defects4J. In total, SOFIX repaired 23 bugs, which are more than previous approaches. We further compared our repaired bugs and our repair templates with previous approaches. Our results show that our additional repair templates and their repair values from Stack Overflow make the improvements.

*Corresponding author

[1]https://archive.org/details/stackexchange

II. Motivating Example

```java
return (...
        cal1.get(Calendar.HOUR) ==
        cal2.get(Calendar.HOUR) && ...);
```

(a) the buggy code

```diff
  return (...
+         cal1.get(Calendar.HOUR_OF_DAY) ==
+             cal2.get(Calendar.HOUR_OF_DAY) &&
-         cal1.get(Calendar.HOUR) ==
-             cal2.get(Calendar.HOUR) && ...);
```

(b) our patch

```java
 Date inDate = null;
...
Calendar cStartOfDate = new GregorianCalendar();
cStartOfDate.set(Calendar.HOUR, 0);
```

(c) the code sample in a question post

```java
Date inDate = inDF.parse("2014-06-05 17:50:50");
...
Calendar cStartOfDate = new GregorianCalendar();
cStartOfDate.set(Calendar.HOUR_OF_DAY, 0);
```

(d) the code sample in an accepted answer post

Fig. 1: The motivating example

In this section, we use a bug from Defects4J [13], the `Lang21` bug, to further illustrate the challenges and benefits of SOFix. Figure 1a shows the buggy code. In a buggy statement, programmers use `Calendar.HOUR` as the argument, when they call the `get` method. As shown in the API document[2], given the input, the `get` method returns the hour of the morning or afternoon. However, the return value is against the intension of programmers, so they replace `Calendar.HOUR` with `Calendar.HOUR_OF_DAY`. With the modified input, the `get` method returns the hour of the day.

SOFix generates the patch as shown in Figure 1b to repair the `Lang21` bug. Defects4J presents manually written patches. We find that our generated patch is identical to its human-written patch. To the best of our knowledge, previous approaches are insufficient to repair this bug. For example, Gao *et al.* [10] repair crash bugs according to existing samples in Stack Overflow. Their approach cannot repair this bug, since the bug does not crash.

Existing approaches typically repair bugs at the granularity of statements. As shown in Figure 1a, in this example, the located faulty statement is lengthy, but only two input values shall be replaced. Early approaches (*e.g.*, [42]) replace buggy statements with statements at other locations. These approaches are insufficient to repair the above bug, since it is unlikely that other locations have the correct but lengthy statement. Recent approaches [44], [17] can repair bugs within statements, but they focus on `if`-statements. As the buggy line of this example is a `return`-statement, the above approaches are insufficient either.

Stack Overflow contains many useful threads that discuss how to repair bugs. In particular, we find a relevant discussion for this example[3]. From the discussion, SOFix identifies the code pair as shown in Figures 1c and 1d. After comparing their ASTs, SOFix extracts the modifications such as updating the values from `Calendar.HOUR` to `Calendar.HOUR_OF_DAY`, deleting a `null` value, and inserting an invocation on the `parse` method. It is infeasible to use the above code samples to repair the bug as Gao *et al.* [10] did, since only one modification is useful. Reusing the above code samples as a whole does not repair the bug in Figure 1a.

[2]https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html
[3]http://stackoverflow.com/questions/24056453

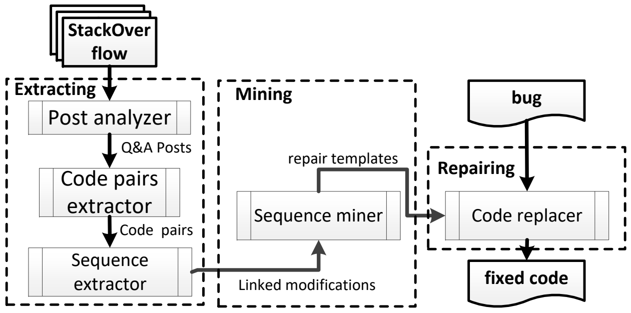

Fig. 2: The overview of SOFix

Instead of the granularity of statements, SOFix repairs bugs at a finer granularity of AST nodes (*e.g.*, variables). From Stack Overflow, SOFix mines a repair pattern. The pattern says that when repairing bugs, a variable can be changed from `Calendar.HOUR` to `Calendar.HOUR_OF_DAY`. Based on this pattern, we implement the *Variable Replacer* template in SOFix, and it repairs the bug in this section with the template. We introduce the mining process in Section III and evaluate our repair templates in Section IV.

III. Approach

Figure 2 shows the overview of SOFix. It has three major steps such as extracting (Section III-A), mining (Section III-B) and repairing (Section III-C).

A. Extracting Stack Overflow

SOFix has three components that sequentially extract repair actions from Stack Overflow.

Algorithm 1 Extracting and Filtering Q&A Posts in Algorithm

```
Input: $PLIST$: a list records all posts.
Output: $QALIST$: a list stores the mapped Q&A posts.
 1: Init for $QLIST$
 2: for each $p \in PLIST$ do
 3:     if $p.type == 1$ then
 4:         $q \leftarrow p$
 5:         if $q.tags.contain($"java"$)$ then
 6:             $QLIST.add(q)$
 7:         end if
 8:     else
 9:         $a \leftarrow p$
10:         search for $q \in QLIST$,
11:         such that
12:         $q.AcId == a.Id$ && $p.PId == a.Id$
13:         do
14:             $QALIST.add(pair < q, a >)$
15:             $QLIST.remove(q)$
16:     end if
17: end for
```

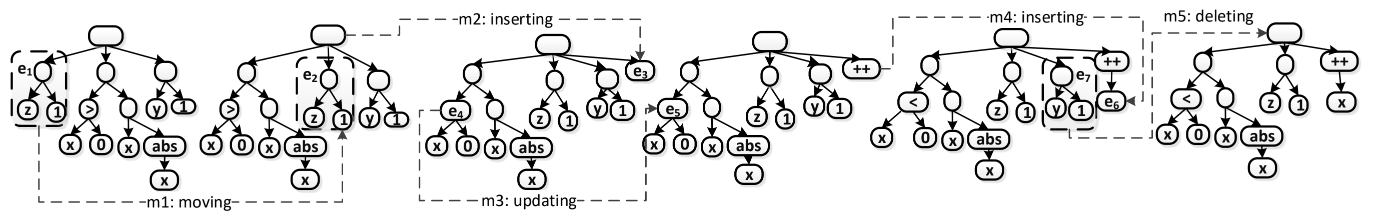

Fig. 3: An example for linked modifications

1. The post analyzer rebuilds the links between questions and their answers. In the archives of Stack Overflow, posts are stored in the format of XML. For example, the following XML snippet includes three posts:

```
// question post
<row Id="4"
     PostTypeId="1"
     AcceptedAnswerId="7"
     AnswerCount="13" .../>
// accepted answer post
<row Id="7"
     PostTypeId="2"
     ParentId="4" .../>
// answer post
<row Id="78"
     PostTypeId="2"
     ParentId="4"
     Score="34" .../>
```

The posts are sorted by their creation dates, so a question can be followed by irrelevant answers. Algorithm 1 shows the procedure to rebuild the links. Line 1 initializes an empty list `QLIST` to store unmapped questions, when it traverses all posts. Line 2 to Line 17 add questions and their accepted answers to the `QAList` list according to the ids. Line 15 removes mapped questions from `QLIST` to reduce the searching effort.

2. The code pairs extractor builds code pairs from each pair of a question and its accepted answer. In particular, from each post, SOFIX extracts code samples, according to the tags such as `<code>` and `<pre>`. Here, a post can have multiple code samples. For example, an answer can rewrite the buggy code, so it is easier to understand. Gao *et al.* [10] use key words (*e.g.*, “instead of” and “change...to...”) to build their sample pairs, but the heuristic is not fully reliable. SOFIX determines code pairs based on their text similarity values, since the buggy and fixed code samples are often similar. The similarity measure is defined as follow:

$$Sim(b,f)=\frac{SimTokenNum(b,f)}{c*Len(b)+(1-c)*Len(f)} \tag{1}$$

In the above equation, b denotes a buggy code sample; f denotes the corresponding fixed code sample; and c is a coefficient. It stands for the weights of token numbers in buggy and fixed code samples when calculating their similarities. $SimTokenNum$ calculates identical tokens between two code samples and Len counts tokens inside a code sample.

3. The modifications sequence miner is built on Spoon [34]. Spoon defines a meta model[4] to denote code elements. The meta model includes three parts: (1) the structural part contains the declarations of the program elements (*e.g.*, interface, class, variable, and method); (2) the code part contains executable Java code (*e.g.*, method bodies); and (3) the reference part models the references to program elements (*e.g.*, a reference to a type). Based on the model, Spoon identifies three types of code elements such as structural elements, code elements and references. Given two ASTs (a_1 and a_2), SOFIX leverages GumTree [9] to generate a modification sequence that transforms a_1 to a_2. Here, we formally define the concepts that are related to modifications:

Definition 1 (Element)**.** *An element is a node or a subtree of an AST.*

For an e element, we use $type(e)$ to denote its type, and $value(e)$ to denote its value. If e is a subtree, we define its type as the type of its root.

Definition 2 (Action)**.** *An action is an atom operation on an element. The atom operations include inserting, updating, moving, and deleting.*

Definition 3 (Modification)**.** *A modification is represented as* $m\langle a,e,e'\rangle$*, where* a *is an action;* e *is the source element; and* e' *is the target element.*

When a is inserting, e is $\varnothing$, and when a is deleting, e' is assigned to $\varnothing$. We use $e_1 \leftarrow parent(e_2)$ or $e_2 \leftarrow child(e_1)$ to denote that e_1 (or the root of e_1 when e_1 is a subtree) is the parent of e_2 (or the root of e_2 when e_2 is a subtree).

As code samples in Stack Overflow are code fragments, Spoon often fails to build correct ASTs from such samples. To handle the problem, as Zhong and Su [56] did, SOFIX adds `import`-statements that import common packages such as `java.util` and `java.io` to the beginning of code samples. In addition, it adds type-declaration and method-declaration statements, if such statements are missing. For a pair of ASTs, SOFIX leverages GumTree [9] to extract a modification sequence. For example, a pair of code samples are as follow:

```
//Buggy Code (left)  //Fixed Code (right)
z = 1;                 if(x<0)
if(x>0)                  x = abs(x);
  x = abs(x);          z = 1;
y = 1;                 x++;
```

The extracted modifications are as follow:

```
//Modifications of Output
1. M1<moving,e1,e2>
2. M2<inserting,-,e3>
3. M3<updating,e4,e5>>
4. M4<inserting,-,e6>
5. M5<deleting,e7,->>
```

[4]http://spoon.gforge.inria.fr/structural_elements.html

TABLE I: The manual inspection subjects

Category	Number	SubE	NodeE	U	I	D	M
P	15	1	14	0.93	0.4	0.33	0.2
M	85	5	80	0.67	1.11	1.52	0.53

Furthermore, SOFIX splits a modification sequence into linked modification sequences, by their modified elements.

Definition 4 (Linked modification)**.** *We consider two modifications ($m_1\langle a, e_1, e_2\rangle$ and $m_2\langle a', e_3, e_4\rangle$) are linked, if $e_2 \leftarrow parent(e_3)$ or $e_2 \leftarrow parent(e_4)$. Special for the case that the two modified elements are nodes, if $parent(e_2) \leftarrow parent(e_3)$ or $parent(e_2) \leftarrow parent(e_4)$, they are linked.*

Figure 3 shows the links of the five modifications. Based on the links, SOFIX builds four linked modification sequences such as $\langle m_1\rangle$, $\langle m_2, m_4\rangle$, $\langle m_3\rangle$, and $\langle m_5\rangle$.

B. Mining Repair Patterns

SOFIX then merges linked modification sequences, if they are isomorphic. We define the isomorphism as follow:

Definition 5 (Modification Isomorphism)**.** *We consider that a modification $m_1\langle a_1, e_1, e_2\rangle$ is isomorphic to another modification $m_2\langle a_2, e_3, e_4\rangle$ if*

1) $a_1 = a_2 = inserting$ *and* $type(e_2) = type(e_4)$ *or*
2) $a_1 = a_2 = updating$ *and* $type(e_1) = type(e_3)$ *or*
3) $a_1 = a_2 = moving$ *and* $type(parent(e_2)) = type(parent(e_4))$ *or*
4) $a_1 = a_2 = deleting$ *and* $type(parent(e_1)) = type(parent(e_3))$

The definition considers action types, modified elements and their parents. For `Updating` and `Inserting` actions, it checks whether modified element types are linked. For `Moving` and `Deleting` actions, it checks whether the types of parent elements are linked.

Definition 6 (Sequence Isomorphism)**.** *A modification sequence $S_1\langle m^1_1, m^1_2, ..., m^1_{n_1}\rangle$ is isomorphic to another modification sequence $S_2\langle m^2_1, m^2_2, ..., m^2_{n_2}\rangle$ if*

1) $n_1 = n_2$ *and*
2) $\forall i \leq n_1$, m^1_i *is isomorphic to* m^2_i

We define the modification sequence isomorphism based on the modification isomorphism. In particular, we require that the lengths of sequences are the same and all the corresponding modifications are isomorphic as defined in Definition 5. Based on the above definitions of isomorphism, SOFIX puts linked modification sequences into categories.

After categories are produced, SOFIX mines a repair pattern from each category. In this paper, a repair pattern is a set of linked modification sequences that repair a type of bugs. However, we find that SOFIX builds many categories, and most of them are redundant or even irrelevant, due to two major reasons. First, a considerable portion of code samples are designed for purposes other than repairing bugs (*e.g.*, illustrating API usages). Second, our underlying tool [9] relies on structural positions to extract modifications, but extracted modifications do not present our desirable semantic mappings.

TABLE II: Mined repair patterns

Id	Pattern Name	Modification Sequence in Repair Pattern
* denotes a node when behind an action, otherwise denotes a parent		
1	InvoReplacer	Update Invocation under *
2	TypeReplacer	Update Type under *
3	VarReplacer	Update Variable under *
4	ArgChanger	Update Invocation under * Insert Variable under Invocation Delete Variable under Invocation
5	ArgAdder	Update Invocation under * Insert Variable under Invocation
6	BinaryOpReplacer	Update BinaryOperator under *
7	ArgRemover	Update Invocation under * Delete * under Invocation
8	BinaryOpInversion	Update BinaryOperator under * Insert BinaryOperator under BinaryOperator Move * from BinaryOperator to the other Move * from BinaryOperator to the other Move * from BinaryOperator to the other Delete BinaryOperator inside BinaryOperator
9	VarToInvo	Insert Invocation under * Delete VariableRead under *
* denotes a subtree when behind an action, otherwise denotes a parent		
10	StateRemover	Delete *
11	ReturnAdder	Insert Return under *
12	IfChecker	Insert If under * Move Block from * to If

We randomly inspected 100 categories to understand how to mine repair patterns. Here, we ignore categories that have fewer than 5 instances. We carefully checked the isomorphic modification sequences and the instances in these 100 categories, and tried to summarize effective and feasible repair patterns. Table I shows the results. For Column "Category", "P" denotes categories with repair patterns, and "M" denotes categories without patterns. Column "Number" lists the number of categories. Column "SubE" lists categories whose modified elements are subtrees. Column "NodeE" lists categories whose modified elements are nodes. Columns "U", "I", "D", and "M" list the averages of updating, inserting, deleting, and moving per modification sequence, respectively.

According to the results of manual inspections, we summarize the following three findings:

1) Most linked sequences with patterns have one or more updating actions.
2) Most linked sequences without patterns have only deleting and moving actions.
3) Linked sequences with multiple deleting and inserting actions seldom show any repair patterns.

Based on the findings, we design heuristics and implement them into SOFIX to automatically filter categories that are unlikely to contain repair patterns. In particular, based on the first and second findings, SOFIX filters categories that contain several modifications but with only deleting or moving actions, and based on the third finding, SOFIX filters categories whose linked sequences contain more than two deleting or inserting actions. Finally, there remains 136 categories and

SOFIX produces a repair pattern from each remaining category.

C. Repairing Bugs

Based on mined repair patterns, SOFIX implements a set of repair templates that modify buggy code. Our derived repair templates define their compatible usage scenarios, and how to synthesize values when generating patches. Here, the search space of template values includes the buggy program, and the instances where repair patterns are mined. Like some existing approaches (*e.g.*, [31]), the current implementation of SOFIX uses repair templates in a one-shot manner, and it does not combine repair templates for more complicated cases.

A mined repair pattern typically has more than five instances. For its modification sequences, we inspected the structures of the modified elements in ASTs, according to the linked modifications and the values of modified elements in instances. Due to the heavy effort, we did not implement repair templates for all repair patterns. Instead, we selected a subset of repair patterns to show the effectiveness of our approach.

First, we discard repair patterns, if we can not derive their usage scenarios and template values. For example, we find that a repair pattern inserts method invocations, but the inserted methods have various programming contexts in different instances. We cannot ensure where to insert a method and what method shall be inserted, so we have to enumerate to insert a invocation of all methods into each suspicious faulty point. However, we believe the enumeration is too expensive and has a low efficiency in repairing a new bug, so we discard that pattern. Second, we discard repair patterns that are combined with several other simple patterns, since our current implementation works in a one-shot manner. We leave these complicated patterns to our future work. Finally, we discard repair patterns that modify large elements (*e.g.*, interface, class, and method), since we focus on fine-grained bug fixes. Moreover, our underlying fault localization tool [4] cannot locate faulty code elements that are larger than statements.

We find that it is feasible to merge some repair patterns, due to their similar semantics. For example, in the Spoon meta model, a string literal, a variable and a field have different types. As a result, we mine separate repair templates for string literals, variables, and fields. However, such repair patterns are quite similar in their semantics (*e.g.*, inserting a string literal or a variable), so we merge them into a single repair template.

In total, we selected the 12 repair patterns as shown in Table II. The first column lists pattern ids. The second column lists pattern names. The last column lists linked modification sequences. For this column, the asterisk (*) denotes changeable elements, which are either modified elements or their parent elements. We next introduce their details:

1. InvoReplacer. This pattern changes a method invocation to another method invocation. We find that the source and target invocations typically have identical parameter types. For example, in an instance of the repair pattern, we find that an invocation of the `getDeclaredMethod(java.lang.String, java.lang.Class)` method is replaced with an invocation of the `getMethod(java.lang.String,java.lang.Class)` method. The two lists of parameter types are identical.

2. TypeReplacer. This pattern changes a type to another one.

3. VarReplacer. This pattern replaces a variable with another variable.

4. ArgChanger, ArgAdder, ArgRemover. These patterns modify an argument of a method invocation as follows:

The *ArgChanger* pattern either moves or updates an argument of a method invocation. We can see that the *ArgChanger* pattern has one modification on the "Invocation", which means the *ArgChanger* pattern changes to invoke another overloading method with a replaced argument in different type.

The *ArgAdder* pattern inserts a variable to a method invocation, and the *ArgRemover* pattern deletes a argument from a method invocation.

5. BinaryOpReplacer. This pattern changes an operator to another one. In a simple instance, $\geq$ is modified to $\leq$.

6. BinaryOpInversion. This pattern changes the priorities of operators. It has six modifications: three modifies on operators and the other three are moving actions. The modifications seem to be complicated, but from source code, they are easy to understand. For example, we find that `a+b-c` is modified to `a+(b-c)` in an instance.

7. VarToInvo. This pattern replaces a variable with a method invocation. For example, in an instance, the original statement is `int sum = total` and the fixed statement is `int sum = CalcTotal()`. It changes the `total` variable to a `CalcTotal()` method invocation.

8. StateRemover, ReturnAdder, IfChecker. These patterns modify subtrees. In particular, the *StateRemover* pattern removes a buggy statement; the *ReturnAdder* pattern inserts a return statement; and the *IfChecker* pattern adds an `if` checker before a buggy code block.

From the above repair patterns, we next derive our repair templates. As mentioned in the beginning of this section, our repair templates define both usage scenarios and template values. Table III shows our derived repair templates. The last column lists the ids of repair patterns in Table II, where the corresponding repair template is derived from. The detailed descriptions of the repair templates are as follows:

1. The BinaryOperator Replacer template.

Usage Scenario: The desirable buggy statement is a `for`-statement, an `if`-statement, or a `while`-statement, and the statement contains at least an operator that is replaced in one or more instances.

Template Value: This repair template searches the instances of the *BinaryOpReplacer* pattern for the replacements of a given operator.

We carefully checked the asterisk (*) element in the isomorphic modification sequences of the *BinaryOpReplacer* pattern, and found that the element has three typical types such as `for`-statement, `if`-statement, and `while`-statement. As a result, we limit the repair template to the three types of statements. For example, here is a buggy statement `if(a≥b){;}`, it is a `if`-statement and it has one $\geq$ operator. When this repair template modifies the buggy statement, it searches instances in

TABLE III: Repair templates in SOFIX

Template Name	Description	Pattern id
BinaryOperator Replacer	For each operator in an IF or FOR or WHILE statement, this template seeks for a compatible operator to replace it.	6
Variable Replacer	This template changes a variable or several identical variables into another compatible ones.	3
Type Replacer	This template modifies a type in a local variable declaration, it seeks for a compatible type to replace the current one.	2
Arguments Adder	This template adds a compatible variable as a new argument to the invocation of an overridden method.	5
Arguments Mover	This template moves an argument to another compatible position in the argument list.	4
Arguments Replacer	This template changes an argument of an overridden method invocation into a compatible variable with distrinct type.	4
Arguments Remover	For each argument of an overridden method invocation, this template deletes it.	7
Invocation Replacer	This template changes an invocation of a method into another method with identical parameter list.	1
BinaryOperator Inversion	For two linked operators, this template reverses the operating priority of them.	8
Variable To Invocation	For each variable read, this template changes it to a non-parameter method invocation with similar name.	9
Return Statement Adder	This template inserts a return statement before or after a statement.	11
Statement Remover	This template deletes a statement.	10
If Checker Adder	This template inserts a IF statement to check null points before a buggy block.	12

the *BinaryOpReplacer* pattern, and find one modification from $\geq$ to $\leq$. Based on this instance, SOFIX modifies the buggy statement to `if(a≤b){;}`. In total, the *BinaryOpReplacer* pattern contains 357 instances that have repair values.

2. The Variable Replacer template.

Usage Scenario: The buggy statement contains at least a variable that is either appear in one or more instances or has replaceable variables of the same type in the buggy program.

Template Value: This template searches the instances of the *VarReplacer* pattern for replacement values. In addition, it searches the buggy program for variables of the same type.

This template is derived from the *VarReplacer* pattern. When leveraging this template to replace a variable, a replacement variable shall be of the same type to avoid compilation errors. If there are several identical variables in a buggy statement, SOFIX can use this repair template to change all the identical variables with another ones. In total, we collected 849 instances with repair values.

3. The Type Replacer template.

Usage Scenario: The buggy statement is a local variable declaration, and the variable type is replaced in instances.

Template Value: This template searches the instances of the *TypeReplacer* pattern for replacement types.

We derive the *TypeReplacer* pattern to this template. The modifications on the return type of a method appear in the *TypeReplacer* pattern, but the corresponding replace template is different from replacing types of variables. Our current implementation considers replacing types of variables, and ignores return types of methods. For this pattern, we extracted 1,087 instances with repair values.

4. The Argument Adder, Argument Mover, Argument Replacer and Argument Remover templates.

Usage Scenario: The buggy statement contains at least one method invocation.

Template Values: The *Argument Mover* template and the *Argument Remover* template do not need any values. The *Argument Adder* and the *Argument Replacer* template searches the buggy program for added or replaced variables.

The four repair templates modify one argument in method invocations. When applying the four repair templates on overridden methods, SOFIX first checks whether the corresponding modifications do not introduce compilation errors. For example, before SOFIX adds an argument to a overridden method `#func()` of a type, it checks whether the buggy program declares a method with one more parameter. If such a method is found (*e.g.*, `#func(`T_1`)` where T_1 donates a type), SOFIX further searches for variables whose types are T_1, and adds them as a new argument one by one. Here, while the *Variable Replacer* template replaces variables with variables of the same type, the *Argument Replacer* template replaces arguments with variables of only different types.

5. The Invocation Replacer template.

Usage Scenario: The buggy statement contains at least one method invocation.

Template Values: This template searches for methods in buggy program with identical parameter list.

This template is derived from the *InvoReplacer* pattern. As described in the *InvoReplacer* pattern, this derived template searches for other methods with identical parameter list in the buggy program to mutate the method invocation.

6. The BinaryOperator inversion template.

Usage Scenario: The buggy statement contains two operators, and one operator is the parent node of the other operator in the AST of the statement.

Template Values: None.

This template is derived from the *BinaryOpInversion* pattern. As it changes only structures, the repair template does not need template values. This template is able to generate a transformation from the left AST to the right one, *i.e.*, $((e_1\ Op_1\ e_2)\ Op_2\ e_3) \rightarrow (e_1\ Op_1\ (e_2\ Op_2\ e_3))$. Alternatively, it is able to generate a transformation from the right AST to the left one, *i.e.*, $(e_1\ Op_1\ (e_2\ Op_2\ e_3)) \rightarrow ((e_1\ Op_1\ e_2)\ Op_2\ e_3)$. As the modification sequences of the two inverse transformations are isomorphic, a single repair template is sufficient to repair both cases.

7. The Variable To Invocation template.

Usage Scenario: The buggy statement contains at least one variable read.

Template Values: This template searches the buggy program for methods whose names are similar to variable names. Here,

TABLE IV: The bugs in the Defects4J benchmark

Project	Defects	Test Cases	Abbreviation
JFreeChart	26	2,205	Chart
Apache Commons Lang	65	2,245	Lang
Apache Commons Math	106	3,602	Math
Joda-Time	27	4,130	Time
Total	224	12,182	-

TABLE V: Overall result

Project	SOFIX	GenProg	xPAR	Nopol	HistoricalFix	ACS
Chart	5	-	-	1	2	2
Lang	4	-	1	3	7	3
Math	13	5	2	1	6	12
Time	1	-	-	-	1	1
Total	23	5	3	5	16	18

SOFIX requires that the type of the variable is identical with the return type of the method, and it searches only methods without parameters.

This template is derived from the *VarToInvo* pattern. After we inspected all the instances in this pattern, we find that most of the replacement methods have similar names to original variables with no parameters. As a result, we restrict the search scope within such methods.

8. The Return Statement Adder and the Statement Remover templates

Usage Scenario: Every buggy statement.

Template Values: The *Return Statement Adder* template fills the added return statement with the default values of the method return type (*e.g.*, `int` for `0`, `boolean` for `true` or `false`, a non-primitive type for `null`).

The *Return Statement Adder* template is derived from the *ReturnAdder* pattern, and the *Statement Remover* template is derived from the *StateRemover* pattern. The two templates modify subtrees. They are simple and do not need to search for template values.

9. The If Checker Adder template.

Usage Scenario: The buggy statement contains at least one variable whose type is non-primitive.

Template Values: None.

This template is derived from the *IfChecker* pattern. We find that in most instances of in this pattern, variables are checked against `null` values. As a result, for each buggy statement, this repair template checks all its variables against `null` values.

IV. EVALUATION

We implemented SOFIX, and conducted evaluations to explore the following research questions:

(RQ1) What is the effectiveness of SOFIX in repairing real-world bugs (Section IV-A)?

(RQ2) Which bugs are repaired by SOFIX, and which bugs are repaired by other approaches (Section IV-B)?

(RQ3) What are the differences between our repair templates and the ones in other approaches (Section IV-C)?

(RQ4) What are the impacts of SOFIX's internal and underlying techniques (Section IV-D)?

RQ1 concerns the overall effectiveness of SOFIX. We use SOFIX to repair the bugs in the Defects4J [13] benchmark. Our results show that SOFIX repaired more bugs (23 in total) than previous approaches [26], [19], [44].

RQ2 concerns the different effectiveness between our approach and previous approaches. By comparing our repaired bugs with the bugs that are repaired by previous approaches, we find that SOFIX is effective in repairing bugs that need repair values from Stack Overflow or bugs that need fined-grained repairs.

RQ3 concerns the comparison between our repair templates and ones in other approaches. We find that six repair templates in SOFIX do not appear in compared approaches. As we mined fine-grained repair templates from Stack Overflow, SOFIX is more effective in repairing bugs than previous approaches.

RQ4 concerns the internal and underlying techniques of SOFIX. Our results highlight the importance of mining from Stack Overflow, since it is infeasible to repair six bugs without such mining. In addition, our results show that there is an urgent requirement for better fault localization techniques, since fault ranks have significant impacts in repair time.

A. RQ1. Overall Effectiveness

1) Setup: We select the Defects4J [13] benchmark, since it is widely used. Sobreira *et al.* [41] present what repair patterns are required to repair bugs in this benchmark.

To explore this research question, we compare SOFIX with five previous approaches. In particular, GenProg [42] is a pioneer approach in automatic program repair. PAR [17] includes more repair templates that are learnt from human patches. Nopol [45] and ACS [44] repair `if`-conditions from hints of documents and code samples. HistoricalFix [19] repairs bugs with manual repair histories. We compared with these approaches, since these approaches are evaluated on the Defects4J [13] benchmark.

Table IV shows the benchmark. The first column lists the four projects in Defects4J. The second and third columns respectively list the numbers of defects and test cases in each project. The last column lists abbreviations for the projects. We did not select bugs from the Closure Compiler project[5], since Defects4J does not provide JUnit test cases for bugs of this project.

For each bug, SOFIX uses GZoltar [4] to its locate faults. GZoltar is a spectra-based fault localization tool. Given a set of statements and test cases, it produces a rank list of suspicious faulty statements. GZoltar has a suspicious threshold to determine whether a statement contains a fault. We set the threshold as 0.01 in our evaluation.

With the support of Astor [28], SOFIX implemented all the repair templates in Table III. For each fault, SOFIX selects its compatible repair templates to synthesize patches, until it enumerates all the suspicious faults or reaches our predefined time limit. Defects4J provides manually written patches. Besides passing test cases, we manually compare synthesized patches with manually written patches. We consider that a bug is correctly fixed, only when a generated patch is identical

[5]https://code.google.com/closure/compiler

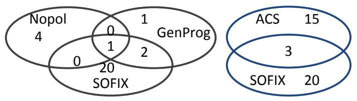

Fig. 4: The comparison results of repaired defects

or semantically equivalent to its manually written patch. The evaluations were conducted on a Ubuntu server with 2.00 GHz Intel Xeon E5-2620 CPU and 16GBs of memory. We set the maximum repair time for each bug as 3 hours.

2) Result: Table V shows that SOFIX repaired 23 bugs in total. Furthermore, Table V lists the results of previous approaches. We find that SOFIX repaired more bugs than any previous approaches. Here, it is worthy mentioning that our results are achieved under more fair settings. For example, HistoricalFix [19] manually located faults, but SOFIX uses GZoltar to locate faults, which can reduce its effectiveness. In addition, SOFIX is a general approach, but some previous approaches (*e.g.*, [44]) repair only `if`-statements.

B. RQ2. Repaired Bugs

1) Setup: We manually compared our repaired bugs with those of previous studies such as Martinez *et al.* [26] and Xiong *et al.* [44]. We did not compare the repaired bugs of HistoricalFix, since Le *et al.* [19] did not present the names of their repaired bugs. We leave this comparison to our future work. Martinez *et al.* [26] also set the maximum time for repairing bugs as 3 hours. Xiong *et al.* [44] set the maximum time as 30 minutes. However, their approach focuses on only `if`-statements, and may not repair more bugs with more time.

2) Result: Figure 4 shows the results. We find that all the approaches typically repaired different bugs. In particular, our repaired bugs involve simple modifications. For example, by replacing a single variable, SOFIX repaired the six bugs such as `Chart11`, `Chart24`, `Lang6`, `Lang59`, `Math5`, and `Math59`. An example is as follow:

```
// patch for Chart11
   PathIterator iterator1 = p1.getPathIterator(null);
-  PathIterator iterator2 = p1.getPathIterator(null);
+  PathIterator iterator2 = p2.getPathIterator(null);
```

Among the six bugs, ACS repaired `Math5`, since only its buggy location is inside an `if`-statement. Nopol did not repair any of the six bugs, since it repaired only `if`-conditions. GenProg repaired `Math5`, since only this program has the correct statement for replacement. PAR has the potential to repair both `Lang6` and `Lang59`, but Le *et al.* [19] report that PAR repaired only one defect in `Lang` project.

By replacing a binary operator, SOFIX repaired `Chart1`, `Math82`, `Math85`, and `Time19`. For example, SOFIX generated the following patch for `Chart1`:

```
// patch for Chart1
-  if (dataset != null) {
+  if (dataset == null) {
       return result;
   }
```

ACS [44] repaired the two bugs. PAR can use its template, changing `if`-predicts with expressions, to repair the two bugs, but only `Chart1` and `Math85` can be repaired in this way.

TABLE VI: The details of repaired bug

Bug Id	Template	Source	Time	Rank	Tries	Psize
Chart1	BinaryOperator Replacer	SO	846	28	514	1
Chart4	If Checker Adder	-	2,224	51	624	2
Chart11	Variable Replacer	P	8	5	31	1
Chart24	Variable Replacer	Both	5	3	22	1
Chart26	If Checker Adder	-	5,324	132	1,556	2
Lang6	Variable Replacer	P	1,812	121	1,213	1
Lang21	Variable Replacer	SO	10	2	19	1
Lang51	Return Statement Adder	-	15	12	64	1
Lang59	Variable Replacer	Both	75	19	209	1
Math2	BinaryOperator Inversion	-	40	1	25	1
Math5	Variable Replacer	P	1	1	1	1
Math33	Arguments Replacer	P	547	20	372	1
Math34	Variable To Invocation	P	2	1	2	1
Math50	Statement Remover	-	155	3	170	4
Math57	Type Replacer	SO	22	2	10	1
Math58	Arguments Remover	-	93	11	92	1
Math59	Variable Replacer	P	178	2	74	1
Math70	Arguments Adder	P	8	1	16	1
Math75	Invocation Replacer	P	1	1	3	1
Math80	BinaryOperator Inversion	-	153	14	175	1
Math82	BinaryOperator Replacer	SO	396	48	623	1
Math85	BinaryOperator Replacer	SO	398	43	452	1
Time19	BinaryOperator Replacer	SO	10,102	313	4,196	1

No previous approaches repaired the `Math80` defect. SOFIX repaired this bug, by reversing its operator priority:

```
// patch for Math80
- int j = 4 * n - 1;
+ int j = 4 * (n - 1);
```

An instance[6] of the corresponding repair pattern is as follow:

```
// buggy code in question post
if (idx - offset % n == 0)
{
    retVal.concat(parts[idx] + "-");
}
```

```
//fixed code in accepted answer post
if ((idx - offset) % n == 0) // added parantheses
{
    retVal += Character.toUpperCase(parts[idx].charAt(0))
    + parts[idx].substring(1) + " ";
}
```

The above code samples show how to repair a similar bug, *i.e.*, changing the operator priority between "-" and "%".

In summary, our results show that SOFIX is effective in repairing bugs that need fined-grained repairs or repair values from Stack Overflow. In addition, SOFIX repaired numbers of new bugs that were not repaired by the compared approaches.

C. RQ3. Repair templates

1) Setup: SOFIX analyzed 31,017,891 posts from Stack Overflow. During mining, we set the threshold as 0.75 and the coefficient `c` as 0.1, for Equation 1. In total, we derived 13 repair templates, as shown in Table III. We compared our repair templates with those of PAR [17] and HistoricalFix [19]. The repair templates in PAR are manually summarized from human-written patches and HistoricalFix uses the templates in previous researches of mutation testing [33], [25] and repair techniques [42], [17].

[6]http://stackoverflow.com/questions/35012637

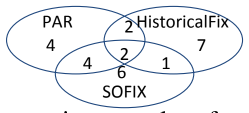

Fig. 5: The comparison results of repair templates

2) Result: Figure 5 shows the results. Here, we refer to finer granularity of repair templates. For example, as HistoricalFix splits the *Expression Adder and Remover* template of PAR into two different templates, we count the *Expression Adder and Remover* template of PAR as two templates. We find that our following repair templates appear in PAR or HistoricalFix:

Our *BinaryOperator Replacer* template is the same as the *Change Infix Expression* template in HistoricalFix. Both templates modify operators. When modifying operators, HistoricalFix tries different operators randomly, but SOFIX searches instances from Stack Overflow. Our *Invocation Replacer* template is the same as the *Method Replacer* template in PAR.

Our *Variable Replacer, Arguments Adder, Arguments Remover, Arguments Replacer* templates are similar to the *Expression Replacer, Parameter Adder and Remover, Parameter Replacer* templates in PAR. The main difference lies in their repair granularity. The granularity of SOFIX is finer, since our above four templates modify variables, but the three templates in PAR modify code at the granularity of expressions. As a result, PAR is less effective to repair the bugs with small faults. As shown in Section II, a buggy statement can have lengthy expressions, but only a small portion of such expressions are faulty. Even if the faulty statement is correctly localized, it is difficult for PAR to repair the statement. In the contrast, SOFIX repaired the bug, since its *Variable Replacer* template repairs finer code elements (*e.g.*, variables).

Our *If Checker Adder* template corresponds to the *Null Pointer Checker* template in PAR. The difference lies in their target code elements. In particular, our *If Checker Adder* template adds `if` checkers to blocks, but the *Null Pointer Checker* template in PAR adds such checkers to statements. For example, suppose that a buggy statement S_1 followed by two associated statements S_2 and S_3. The *Null Pointer Checker* template adds checkers like `if(){`S_1`}` S_2 S_3, but our *If Checker Adder* template adds checkers like `if(){`S_1 S_2 S_3`}`.

Our remaining 6 templates do not appear in the referred two approaches. To the best of our knowledge, our *BinaryOperator Inversion* and *Variable To Invocation* never appear in previous researches. As more fine-grained repair templates are mined from Stack Overflow, for the Defects4J benchmark, SOFIX repaired more bugs than previous approaches did.

D. RQ4. Internal and Underlying Techniques

1) Setup: When SOFIX repaired bugs, we recorded its internal data. For example, we recorded how SOFIX synthesized values for repair templates, and the time of repairing each bug. We manually inspected the data to present our results.

2) Result: Table VI shows our results. The second column lists templates that repaired the corresponding bugs. Column "Source" shows source of template values. In particular, "SO" denotes that SOFIX obtained the correct values from Stack Overflow; "P" denotes that SOFIX obtained the correct values from the buggy program under repairing; and "Both" denotes that SOFIX could obtain the correct values from both sources. Column "Time" lists repair time in seconds. Here, we ignore the time of fault localizations to make our results consistent with the evaluations in existing papers [26], [19], [44]. Column "Rank" lists rankings of correct faulty statements. Column "Tries" denotes tried times before bugs were repaired. Column "Psize" lists our generated patch sizes, which are the numbers of modified code lines.

Column "Source" shows that in six repaired bugs, SOFIX could only extract correct repair values from Stack Overflow. The results highlight the importance of SOFIX, since it is infeasible to repair the six bugs without mining Stack Overflow.

The repair time is largely in proportion to the rankings of correct faulty statements. It took less than a minute for SOFIX to repair bugs, if the underlying fault localization technique [4] correctly ranks faulty statements at the top. In the contrast, if the underlying fault localization technique fails to rank faults at the top, it took much longer for SOFIX to repair bugs. For example, as the correct fault of `Time19` is ranked as the 313rd suspicious statement, it took more than two hours to repair this bug. Indeed, as the faulty statement of `Lang26` is too low, SOFIX failed to repair the bug within time limit. Although our repair templates are able to repair the bug, it took more than four hours for SOFIX to repair the bug. From our results, it is desirable for researchers to propose more effective fault localization approaches.

As our approach is based on one-shot manner and fine-grained repair templates, our generated correct patches are typically small. As shown in Column "Psize", most of our repaired bugs involve only one-line modifications. `Chart4`, `Chart26`, and `Chart50` are repaired by those repair templates that modify subtrees (*e.g.*, If Checker Adder), so their patches modify more than one code line.

Debroy and Wong [7] define three types of bugs:

- L1: programs with a single fault on a single statement.
- L2: programs with a single fault on multiple statements.
- L3: programs with multiple faults.

In this paper, we focus on L1 bugs, but leave L2 and L3 bugs for future work. Our results show that if faults are not correctly located, it takes much more time to repair bugs. DiGiuseppe and Jones [8] show that existing approaches are effective to locate only a single fault of $L2$ and $L3$ bugs. We further discuss this issue in Section V.

E. Threats to Validity

The threats to internal validity include the limitation of the underlying tools, since Spoon can ignore code elements and Gumtree can ignore feasible edit actions. The threat can be reduced with more advanced tools. The threats to internal validity also include the manual process to derive repair templates. To reduce the threat, we carefully inspect mined patterns, and it can be further reduced with introducing more researchers in the inspection. The threats to external validity includes our chosen Defects4J benchmark and Stack Overflow repository. Although Defects4J is widely used, our effectiveness can be reduced in repairing other bugs. The threat can be reduced by repairing more bugs from other sources. In addition, we can extract more bug repair data in other repositories to mitigate the dependencies on Stack Overflow.

V. Discussion and Future Work

Fixing bugs with multiple faults. We notice that a fault can appear in multiple locations. For example, repairing `Chart7` needs replacing `minMiddleIndex` with `maxMiddleIndex` in two statements. As single faults are repetitive, it can be feasible to repair these bugs, if their faults are located. DiGiuseppe and Jones [8] complain that existing fault localization techniques are insufficient to locate multiple faulty locations. In future work, we plan to explore how to repair multiple faults.

Fully automate SOFix. After SOFix mines repair patterns, we have to manually select useful ones. In addition, we have to manually implement repair templates according to mined repair patterns. The manual process is tedious and can lose useful patterns. Although we introduce heuristic filter rules, these rules are not fully reliable. In future work, we plan to explore more advanced techniques to full automation. For example, it is a hot research topic to mine specifications [54], [37]. It may be feasible to use mined specifications to determine useful repair patterns.

Exploring missing issues in our evaluations. Our evaluations do not fully explore some issues. First, we inspected 100 mined categories, but inspecting more categories can lead to more findings. Second, we empirically set coefficient `c` as 0.1 and the threshold as 0.75, but did not explore the impacts of other values. Third, we did not compare our repaired bugs with the repaired bugs of HistoricalFix [19], due to the effort of re-implementing the tool. Finally, we manually determine correct patches, which can lead to errors. In our future work, we plan to handle these issues. For example, Liu *et al.* [22] compare test coverage to determinate correct patches. While we use passed test cases and manual inspection to determine correct patches, their criterion may be useful to reduce the manual bias in this process.

VI. Related work

Automatic Program Repair. Given a buggy program, automatic program repair searches for a source-level patch that repairs the bug [42]. Various approaches are proposed to efficiently search for patches [43], [35], [30], [16], [23], [24], [27]. BugFix [14] employs the apriori algorithm to rank previous fix. SemFix [32] repairs bugs in assignments and conditions. Liu *et al.* [20] learn from bug reports to fix bug. Rolim *et al.* [38] and Le Goues *et al.* [18] repair bugs based on examples. Saha *et al.* [39] rank patches with more repair templates and algorithms. Chen *et al.* [5] leverage learnt contracts to fix bugs. Researchers still hold controversial opinions about the effectiveness of various approaches. For example, Qi *et al.* [35] show that RSRepair is more effective than GenProg, while Smith *et al.* [40] observe the opposite. Le Goues *et al.* [11] prepare a benchmark for the follow-up research. Recent studies [48], [49] show that better test cases can improve the effects of bug repair. Yang *et al.* [47] show that the suspiciousness-first algorithm works better in parallel repair and patch diversity, comparing with the rank-first algorithm. Hassan and Wang [12] repair build scripts with fixing histories. Our approach mines repair templates from Stack Overflow, complementing existing approaches with more repair templates.

Empirical studies on bugs and fixes. Many researchers [15], [50], [51], [52], [53] manually inspect code changes and commit messages to understand bug fixes. Barr *et al.* [2] report that 11% bug fixes can be fully reconstituted from existing code. Martinez *et al.* [29] report that at the line granularity, 3% to 17% bug fixes are temporal redundancy. It is feasible to fix new bugs with historical sources. With the support of an advanced tool [58], Zhong and Meng [55] conduct a more indepth study on reusing hints from past fixes. Our approach mines repair templates from the other source than past fixes.

Analyzing online forums. Online forums such as Stack Overflow provide a rich source for analysis. Cong *et al.* [6] proposed a sequential patterns based classification method to detect question-answer pairs in a forum thread. Yang *et al.* [46] use an adaptive feature-based matrix factorization framework to recommend relevant posts. Bhatia *et al.* [3] proposed a model for online thread retrieval based on inference networks that utilized the structural properties of forum threads. Albaham *et al.* [1] adopted several voting techniques that had been applied in ranking aggregates tasks such as blog distillation and expert finding. Our approach analyzes forum posts from a different angle, mining repair templates.

VII. Conclusion

Automatic program repair has been a hot research topic in the software engineering community, but existing approaches are less effective in repairing bugs, partially due to their limited repair templates. Meanwhile, when programmers repair bugs, they often search Stack Overflow to learn how to repair bugs. This observation leads to our work that mines repair templates from Stack Overflow. In this paper, we propose SOFix that mines fine-grained repair patterns from Stack Overflow. Based on such patterns, we derive 13 repair templates. We conduct an evaluation on the Defects4J benchmark. Our results show that 23 bugs are repaired, which shows the effectiveness of our approach. We further analyze repaired bugs in our work and previous papers, and our findings highlight the importance of mining repair templates from Stack Overflow.

Acknowledgments

We appreciate the reviewers for their constructive comments. This work is sponsored by the 973 Program in China No. 2015CB352203, the National Nature Science Foundation of China No. 61572313 and No. 61472242, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305.

References

[1] A. T. Albaham and N. Salim. Adapting voting techniques for online forum thread retrieval. In *Prod. 1st AMLTA*, pages 439–448, 2012.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proc. FSE*, pages 306–317, 2014.

[3] S. Bhatia and P. Mitra. Adopting inference networks for online thread retrieval. In *Prod. 24th AAAI*, 2010.

[4] J. Campos, A. Perez, and A. Rui. Gzoltar: an eclipse plug-in for testing and debugging. In *Proc. ASE*, pages 378–381, 2012.

[5] L. Chen, Y. Pei, and C. A. Furia. Contract-based program repair without the contracts. In *Proc. ASE*, pages 637–647, 2017.

[6] G. Cong, L. Wang, C. Lin, Y. Song, and Y. Sun. Finding question-answer pairs from online forums. In *Prod. 31st SIGIR*, pages 467–474, 2008.

[7] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proc. ICSE*, pages 65–74, 2010.

[8] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proc. ISSTA*, pages 210–220, 2011.

[9] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proc. ASE*, pages 313–324, 2014.

[10] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proc. of ASE*, 2015.

[11] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[12] F. Hassan and X. Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proc. ICSE*, 2018.

[13] D. Jalali and M. D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proc. ISSTA*, pages 437–440, 2014.

[14] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Proc. ICPC*, pages 70–79, 2009.

[15] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proc. PLDI*, pages 77–87, 2012.

[16] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *Proc. ASE*, pages 295–306, 2015.

[17] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. ICSE*, pages 802–811, 2013.

[18] X. B. D. Le, D. H. Chu, D. Lo, C. L. Goues, and W. Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proc. FSE*, pages 593–604, 2017.

[19] X. B. D. Le, D. Lo, and C. L. Goues. History driven program repair. In *Proc. SANER*, pages 213–224, 2016.

[20] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2fix: Automatically generating bug fixes from bug reports. In *Proc. ICST*, pages 282–291, 2013.

[21] X. Liu, B. Shen, H. Zhong, and J. Zhu. Expsol: Recommending online threads for exception-related bug reports. In *Proc. APSEC*, pages 25–32, 2016.

[22] X. Liu, M. Zeng, Y. Xiong, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. *Proc. ICSE*, 2018.

[23] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proc. FSE*, pages 166–178, 2015.

[24] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proc. POPL*, pages 298–312, 2016.

[25] Y. S. Ma, Y. S. Ma, and Y. R. Kwon. The class-level mutants of mujava. In *International Workshop on Automation of Software Test*, 2006.

[26] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Springer Empirical Software Engineering*, 2016.

[27] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[28] M. Martinez and M. Monperrus. ASTOR: A program repair library for Java. In *Proc. ISSTA*, pages 441–444, 2016.

[29] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proc. ICSE*, pages 492–495, 2014.

[30] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proc. ICSE*, pages 448–458, 2015.

[31] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proc. ICSE*, pages 691–701, 2016.

[32] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proc. ICSE*, pages 772–781, 2013.

[33] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *Acm Transactions on Software Engineering & Methodology*, 1996.

[34] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[35] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proc. ICSE*, pages 254–265, 2014.

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

[37] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

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

[39] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *Proc. ASE*, pages 648–659, 2017.

[40] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proc. FSE*, pages 532–543, 2015.

[41] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proc. SANER*, 2018.

[42] Weimer, Westley, Nguyen, ThanhVu, G. Le, Claire, Forrest, and Stephanie. Automatically finding patches using genetic programming. In *Proc. ICSE*, pages 364–374, 2009.

[43] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proc. ASE*, pages 356–366, 2013.

[44] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *Proc. ICSE*, 2017.

[45] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.

[46] D. Yang, M. Piergallini, I. Howley, and C. Rose. Forum thread recommendation for massive open online courses. In *Prod. 7th ICEDM*, 2014.

[47] D. Yang, Y. Qi, and X. Mao. An empirical study on the usage of fault localization in automated program repair. In *Proc. ICSME*, pages 504–508, 2017.

[48] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proc. ESEC/FSE*, pages 831–841, 2017.

[49] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering*, (8):1–32, 2017.

[50] Z. Yin, X. Ma, J. Zheng, S. Pasupathy, S. Pasupathy, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proc. SOSP*, pages 159–172, 2011.

[51] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. FSE*, pages 26–36, 2011.

[52] F. Yu, H. Zhong, and B. Shen. How do programmers maintain concurrent code. In *Proc. APSEC*, 2017.

[53] R. Yue, N. Meng, and Q. Wang. A characterization study of repeated bug fixes. In *Proc. ICSME*, pages 422–432, 2017.

[54] H. Zhong and H. Mei. An empirical study on API usages. *IEEE Transaction on Software Engineering*, 2018.

[55] H. Zhong and N. Meng. Towards reusing hints from past fixes: An exploratory study on thousands of real samples. *Empirical Software Engineering*, 2018.

[56] H. Zhong and Z. Su. Detecting API documentation errors. In *Proc. OOPSLA*, pages 803–816, 2013.

[57] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. ICSE*, pages 913–923, 2015.

[58] H. Zhong and X. Wang. Boosting complete-code tool for partial program. In *Proc. ASE*, pages 671–681, 2017.