The Journal of Systems and Software 156 (2019) 100–112

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

LoopFix: an approach to automatic repair of buggy loops

Weichao Wang, Zhaopeng Meng, Zan Wang*, Shuang Liu, Jianye Hao

College of Intelligence and Computing, Tianjin University, China

ARTICLE INFO

Article history:
Received 27 February 2018
Revised 16 April 2019
Accepted 18 June 2019
Available online 19 June 2019

Keywords:
Automatic repair
Test suite based
Buggy loop
Symbolic execution
Program synthesis

ABSTRACT

Bugs may be present in various places in a program. Bugs contained in loops (hereafter buggy loops) are challenging to fix using existing bug fixing techniques. In this work, we propose *LoopFix*, an approach that aims to repair buggy loops automatically. *LoopFix* takes a program and a test suite including at least one failed test case as inputs, and generates a patch to fix the bug. *LoopFix* first leverages on existing bug localization techniques to obtain ranked buggy statements. After that, *LoopFix* exploits a component based program synthesis approach to synthesize a patch based on the runtime information obtained through symbolic execution. Finally, *LoopFix* validates the patch with the given test suite. We have implemented *LoopFix* in a prototype tool and compared the performance of *LoopFix* with Angelix, S3 and JFix on three widely adopted datasets, i.e., IntroClass, Defects4J and Loops. Our experiments show that *LoopFix* fixes about 30% of buggy loops and performs more effective than the other tools on buggy loops.

1. Introduction

Debugging is one of the most time-consuming phases in software development life cycle. There have been significant efforts involved in developing automatic program repair techniques in the past decade to reduce the software development life cycle. Program repair techniques can be classified into two classes: search based approaches (Arcuri and Yao, 2008; Le Goues et al., 2012; Goues et al., 2012; Perkins et al., 2009; Kim et al., 2013; Weimer et al., 2013; Qi et al., 2014) and constraint solving based approaches (Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016; DeMarco et al., 2014; Xuan et al., 2017; Le et al., 2017b; Le et al., 2017a; Ke et al., 2015; Long and Rinard, 2015; Tan and Roychoudhury, 2015).

Bugs may be present in various places in a program, including loops. For example, reports show that there are at least 331 loop bugs in Eclipse (Pan et al., 2009), 235 loop bugs in MegaMek (Li et al., 2006) and 137 loop bugs in ArgoUML (Le Goues et al., 2013). While existing bug fixing techniques have been shown to be effective in fixing many bugs, bugs contained in loops (hereafter buggy loops) are proven to be challenging. Repairing buggy loops is more challenging than repairing bugs in other places, due to the complexity involved in loop structures. Given a loop, bugs may reside either in the loop condition or in the loop body and a buggy in the loop condition or the loop body may be executed repeatedly. Furthermore, constraint-solving based repairing methods are by design ineffective due to its limitation in handling loops.

In this work, we propose an approach called *LoopFix* which aims to automatically fix buggy loops. *LoopFix* takes a program and a test suite including at least one failed test case as inputs, and generates a patch to fix the bug. Same as existing program repair methods (Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016; DeMarco et al., 2014; Xuan et al., 2017; Le et al., 2017b; Le et al., 2017a), *LoopFix* first leverages on existing bug localization techniques to obtain a list of ranked buggy statements. That is, *LoopFix* computes the suspiciousness score of each statement to identify potential bug locations using the Tarantula technique (Jones and Harrold, 2005). Given the most suspicious statement, we then classify a bug in a loop structure into three categories, i.e., bugs in the loop condition only, bugs in the loop body only and bugs in both the loop condition and the body. Then, *LoopFix* then employs symbolic execution techniques (Cadar and Sen, 2011) to collect runtime ⟨*key, value*⟩ pairs as oracles. For each type of bugs, using the collected runtime ⟨*key, value*⟩ pairs as inputs, we employ oracle guided component-based program synthesis approach (Jha et al., 2010) to generate repair constraints and then synthesize a patch by solving those constraints. Finally, the provided test suit is re-executed to validate the patched program.

We have implemented *LoopFix* in a prototype tool supporting Java programs and compared the performance of *LoopFix* with Angelix (Mechtaev et al., 2016), S3 (Le et al., 2017b) and JFix (Le et al., 2017a) on three widely adopted datasets, i.e., IntroClass (Le Goues et al., 2015), Defect4J (Just et al., 2014) and Loops (Competition on software verification, 2016). In total, we have a set

* Corresponding author. College of Intelligence and Computing, Tianjin University, No. 135 Yaguan Road, Jinnan District, Tianjin, China.
E-mail address: wangzan@tju.edu.cn (Z. Wang).

https://doi.org/10.1016/j.jss.2019.06.076
0164-1212/

of 171 programs containing buggy loops. We systematically apply *LoopFix* to these programs in order to answer the following two research questions.

1. How effective is *LoopFix* in fixing bugs in loop conditions and loop bodies; and
2. What is the quality of the fix patches generated by *LoopFix*.

The evaluation results show that *LoopFix* successfully fixes about 30% of the buggy loops, i.e., 35.71% for those bugs residing in the loop condition only, 27.27% for those residing in the loop body only and 30.00% for those residing in both the condition and the body. *LoopFix* performs more effective than the other tools on buggy loops. Regarding the quality of the *LoopFix* generated fix patches, we manually examine the generated fixes and find that those successful fixes are functionally equivalent to the manual patches written by experienced programmers.

The main contributions of this work are listed as follows.

- We classify bugs in loop structures into three types and propose a novel way of defining the ⟨*key, value*⟩ pairs used for patch synthesis.
- We demonstrate the effectiveness of *LoopFix* by evaluating it with three well-known benchmarks, and results show that nearly 30% loop bugs are identified and repaired successfully.
- We compare the performance of *LoopFix* with Angelix (Mechtaev et al., 2016), S3 (Le et al., 2017b) and JFix (Le et al., 2017a). *LoopFix* performs more effective than the other tools on buggy loops.
- We demonstrate that the patches synthesized by *LoopFix* are functionally equivalent to those patches provided by experienced programmers.

The remainder of this paper is organized as follows. Section 2 describes the background of our approach. Section 3 introduces the details of our approach. Selection 4 presents the empirical evaluations of *LoopFix* with well-known benchmarks consisting of real bugs. We discuss threat-to-validity in Section 5. Section 6 lists the related work. Finally, we conclude our work in Section 7.

2. Background

2.1. Loop structure and buggy loops

A loop is a control flow which repeatedly executes a block of statements. Loops can be represented in three different structures, i.e., *for, while* and $do-while$. The condition statement in a loop structure controls the conditional execution of the loop body, and is presented with a Boolean expression. Each time before executing the loop body, the loop condition must be evaluated first. The program jumps into the loop body if the loop condition is evaluated to true. This process repeats until the loop condition is evaluated to false. Different loop structures can be transformed into each other as shown in Fig. 1. For simplicity, we focus on the *while* loop in the following.

Loop conditions can be further classified into two categories, i.e., symbolic conditions and constant conditions. The first type is the symbolic condition represented as a symbolic expression, which is hard to predict the number of loop execution in advance. For instance, we cannot predict the number of loop execution for the loop condition $while(x > 0)$ before execution. The second type is the constant condition represented as a constant expression, which allows us to predict the number of loop execution. Considering the loop condition $for(int \quad i = 1; i <= 100; i++)$ as an example, we know directly that it will be executed 100 times without break.

The bugs in buggy loop programs can be classified into three categories, i.e., bugs residing only in loop condition statements, bugs only in loop bodies and bugs in both loop condition and body. Let us take a simple integer summing (from 1 to 100) program as an example. Fig. 2 shows three buggy implementations, which corresponds to the tree types of loop bugs that we defined. In the first case, the bug resides in the loop condition, where $while(i < 100)$ fails to add the last number 100. In the second case, the bug resides in the loop body, where the statement $sum = i$ fails to conduct addition operation. In the last case, both of the above bugs co-exist.

2.2. Test suite based program repair

Test suite based program repair (Arcuri and Yao, 2008; Le Goues et al., 2012; Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016) leverages test suites to fix program bugs. The general idea is to search for a patch such that the patched program passes all test cases in the test suite. These approaches consist of three major processes, i.e., fault localization, patch generation and patch verification, as shown in Fig. 3. In the first process, we compute the suspiciousness score of each statement based on the program spectrum, and obtain a ranked list of suspicious statements. There are two kinds of approaches to generate patches in the second process, i.e., search based approaches (Arcuri and Yao, 2008; Le Goues et al., 2012; Goues et al., 2012; Perkins et al., 2009; Kim et al., 2013; Weimer et al., 2013; Qi et al., 2014) and constraint based approaches (Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016; DeMarco et al., 2014; Xuan et al., 2017; Le et al., 2017b; Le et al., 2017a; Ke et al., 2015; Long and Rinard, 2015; Tan and Roychoudhury, 2015). Search based approaches generate and evaluate a large number of mutated versions of the buggy program using pre-defined mutation operators, from which the patch is selected. Constraint based approaches transform the patch generation problem into the constraint solving problem where the constraints are obtained from the buggy program. In the third process, the whole test suite is re-executed to ensure that the patched program can make all test cases pass.

2.3. Oracle guided component based program synthesis

Oracle guided component based program synthesis (Jha et al., 2010) aims at synthesizing a program f, which satisfies a given set of input-output pairs, with a given set of base components. The input-output pairs act as the specification which the desired program should adhere to. For example, for an input-output pair $\langle \alpha, \beta \rangle$, given the input α, the synthesized program should output β. Moreover, a given set of base components $\{f_1, f_2, \ldots, f_N\}$ are needed as building blocks in the synthesized program f. The base components are domain specific and are selected according to the application. In this work, we provide a set of basic components, which consists of four levels as shown in Table 1. These are the same ones used in previous works (Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016; DeMarco et al., 2014; Xuan et al., 2017). The synthesis procedure starts from component level 0 and gradually goes to level 3 to adapt to the complexity of the program. If sufficient base components are given, the synthesis process can be converted to the problem of finding the locations where each

Table 1
The level of components Nguyen et al. (2013).

Level	Component
0	Constant
1	>, <, ==, !=
2	+, -
3	and, or, not

	template	example

for	***for***$(initialization;$
	$condition; step)$
	{
	$body;$
	}
		int $sum = 0;$
		for(int $i = 1; i <= 100; i++)$
		{
		$sum += i;$
		}

while	$initialization;$
	while$(condition)$
	{
	$body;$
	$step;$
	}
		int $sum = 0;$
		int $i = 1;$
		while$(i <= 100)$
		{
		$sum += i;$
		$i++;$
		}

do-while	$initialization;$
	if$(condition)$
	{
	do
	{
	$body;$
	$step;$
	}
	while$(condition)$
	}
		int $sum = 0;$
		int $i = 1;$
		if$(i <= 100)$
		{
		do
		{
		$sum += i;$
		$i++;$
		}
		while$(i <= 100)$
		}

Fig. 1. Three types of loop structures.

type of buggy loops	example

loop condition bugs	***int*** $sum = 0;$
	int $i = 1;$
	while$(i < 100)$ $//fix$: ***while***$(i <= 100)$
	{
	$sum += i;$
	$i++;$
	}

loop body bugs	***int*** $sum = 0;$
	int $i = 1;$
	while$(i <= 100)$
	{
	$sum = i; //fix : sum += i;$
	$i++;$
	}

loop condition and bogy bugs	***int*** $sum = 0;$
	int $i = 1;$
	while$(i < 100)//fix$: ***while***$(i <= 100)$
	{
	$sum = i; //fix : sum += i;$
	$i++;$
	}

Fig. 2. Three types of buggy loops.

component appears in the desired program. The input-output pairs can be regarded as the constraint over the components' locations, which can be solved by a SMT solver. If there exists a solution, a program can be constructed from the values of location variables. The detailed method is explained in the following paragraphs.

For the ith component f_i, we use $\overrightarrow{x_i}$ to denote its input and r_i to denote its output. We use $Q := \cup_{i=1}^{N} \overrightarrow{x_i}$ to denote the set of all input variables from all components and $R := \cup_{i=1}^{N} \{r_i\}$ to denote the set of output variables from all components. For the desired program f, we use $\overrightarrow{x}$ to denote the input variables and r to denote the output variable. Based on the above definitions, we can further define the set of location variables as $L := \{l_x | x \in Q \bigcup R \bigcup \overrightarrow{x} \bigcup \{r\}\}$, where l_x denotes the location in which variable x is defined. The location variables have to satisfy the following constraints.

If the location variables are obtained from the SMT solver, the synthesized program can be constructed with these location values according to the process shown in Fig. 4. From Line 0 to Line $|\overrightarrow{x}| - 1$, each part of the input $\overrightarrow{x}$ is defined. From Line $|\overrightarrow{x}|$ to

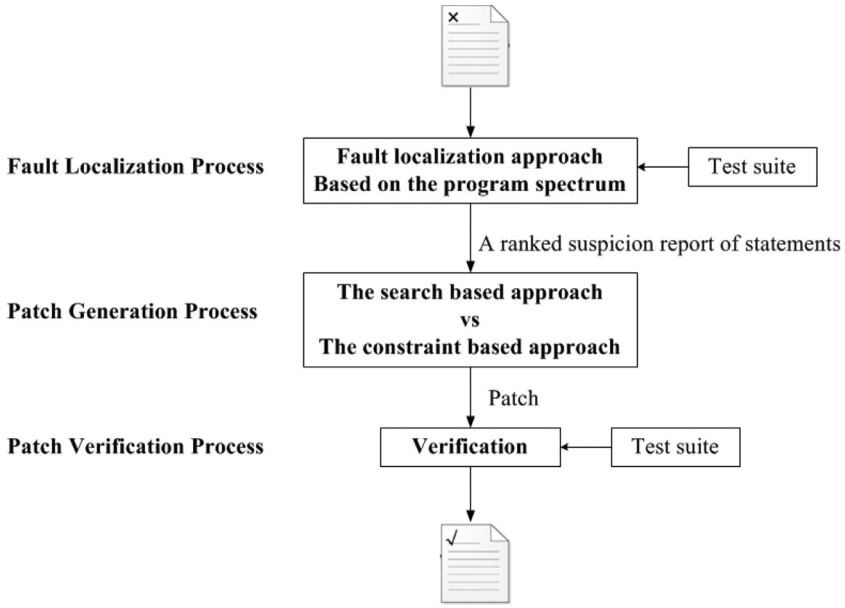

Fig. 3. Test suite based program repair.

Input: $\overrightarrow{\mathcal{X}}$

Output: r

Program($\overrightarrow{\mathcal{X}}$)

{

/* define the input $\overrightarrow{\mathcal{X}}$. */

Line 0:

...

Line $|\overrightarrow{\mathcal{X}}| - 1$:

/* describe the body of program. */

/* $\pi_1, \pi_2, \ldots, \pi_N$ is a permutation of $1, 2, \ldots, N$. */

Line $|\overrightarrow{\mathcal{X}}|$: $r_{\pi_1} = f_{\pi_1}(\overrightarrow{\mathcal{X}}_{\pi_1})$

...

Line $M-1$: $r_{\pi_N} = f_{\pi_N}(\overrightarrow{\mathcal{X}}_{\pi_N})$

Line M: **return** r_{π_N}

}

Fig. 4. The process of synthesizing program from location variables.

Line $|\overrightarrow{\mathcal{X}}| + N - 1$, the synthesized program is shown completely. In Line $|\overrightarrow{\mathcal{X}}| + N$, the synthesized program returns the output r. For the sake of brevity, let $M := |\overrightarrow{\mathcal{X}}| + N$.

$$\psi_{wfp}(L, Q, R) \stackrel{def}{=} \wedge_{x \in Q}(0 \leqslant l_x < M) \wedge_{x \in R}(|\overrightarrow{\mathcal{X}}| \leqslant l_x < M) \wedge \psi_{cons}(L, R) \wedge \psi_{acyc}(L, Q, R) \quad (1)$$

$$\psi_{cons}(L, R) \stackrel{def}{=} \wedge_{x,y \in R, x \neq y}(l_x \neq l_y) \quad (2)$$

$$\psi_{acyc}(L, Q, R) \stackrel{def}{=} \wedge_{i=1}^{N} \wedge_{x \in \overrightarrow{\mathcal{X}}_i, y = r_i}(l_x < l_y) \quad (3)$$

Constraint (2) requires that there is only one component in each line. Constraint (3) encodes that inputs of each component are defined before they are used. Constraint (1) guarantees that f is a well-formed program.

$$\phi_{func}(L, \overrightarrow{\mathcal{X}}, \{r\}) \stackrel{def}{=} \exists Q, R \quad \psi_{wfp}(L, Q, R) \wedge \phi_{lib}(Q, R) \wedge \psi_{conn}(L, \overrightarrow{\mathcal{X}}, \{r\}, Q, R) \quad (4)$$

$$\phi_{lib}(Q, R) \stackrel{def}{=} \wedge_{i=1}^{N} \phi_i(\vec{\chi_i}, \{r_i\}) \quad (5)$$

$$\psi_{conn}(L, \vec{\chi}, \{r\}, Q, R) \stackrel{def}{=} \wedge_{x,y \in Q \cup R \cup \vec{\chi} \cup \{r\}} (l_x = l_y \rightarrow x = y) \quad (6)$$

Constraints (4) consists constraints (5) and (6). Constraint (5) represents the semantic of components that relates the inputs and output of each component. Constraint (6) represents the dataflow semantics that matches the inputs and output of the different components and the inputs and output of the overall program with each other, in accordance with values of location variables.

$$Behave_E(L) \stackrel{def}{=} \wedge_{(\alpha_j, \beta_j \in E)} \phi_{func}(L, \alpha_j, \beta_j) \quad (7)$$

Finally, given a set E of input-output pairs $\{(\alpha_j, \beta_j)\}_{j=1,\ldots,N}$, constraint (7) dictates that, when the input is α_j, the output should be β_j. Given a solution L to constraint (7), we can generate a program that satisfies all the input-output pairs.

3. The LoopFix approach

In this section, we introduce the details of *LoopFix*. *LoopFix* consists of three major processes, i.e., fault localization, patch generation and patch verification. Firstly, *LoopFix* performs fault localization to compute the suspiciousness score of each statement in the loop. Then, *LoopFix* starts synthesizing patches for each suspicious statement based on their suspiciousness scores. An overview of our approach *LoopFix* is shown in Algorithm 1. We distinguish

Algorithm 1: The overall algorithm of *LoopFix*.

```
   Input: P: The buggy loop program, T: A test suite, RS:
          A ranked list of suspicious statements
   Output: r: a patch for P
1  foreach s in RS do
2      if s is the loop condition then
3          r = RepairLoopCondition(P, T, s);
4          if r! = NULL&&Verify(r) == true then
5              return r;
6          else
7              r = RepairLoopConditionAndBody(P, T, s);
8              if r! = NULL&&Verify(r) == true then
9                  return r;
10     else if s is the loop body then
11         r = RepairLoopBody(P, T, s);
12         if r! = NULL&&Verify(r) == true then
13             return r;
14         else
15             r = RepairLoopConditionAndBody(P, T, s);
16             if r! = NULL&&Verify(r) == true then
17                 return r;
```

three types of loop bugs and propose different patch synthesizing algorithms accordingly. If the suspicious statement is a loop condition, *LoopFix* tries to repair loop condition (Line 3) first by calling the method *RepairLoopCondition* defined in Algorithm 2. If a patch is generated and passes all the test cases in the given test suite T (Line 4), the patch is returned and *LoopFix* terminates. If no patch passing all test cases is generated, the loop body must be fixed as well. Thus, *LoopFix* tries to repair loop condition and body

Algorithm 2: RepairLoopCondition.

```
   Input: P: a buggy program, T: a test suite, s: the
          suspicious statement
   Output: r: a patch for P
1  specification = NULL;
2  key[] = NULL;
3  answer = NULL;
4  foreach t ∈ T do
5      while true do
6          key[] = GetReachableValues();
7          answer = GetLoopReturn();
8          if answer! = t.output then
9              specification ← ⟨key, true⟩;
10         else
11             specification ← ⟨key, false⟩;
12             break;
13 r = Repair(P, specification);
14 if r! = NULL then
15     return r;
```

Algorithm 3: Repair

```
   Input: P: a buggy program, specification
   Output: r: a patch for P
1  constraint = GenerateRepairConstraint(P, specification);
2  level = 0;
3  r = Synthesize(constraint, level);
4  while r == NULL && level ≤ MAX_LEVEL do
5      level++;
6      r = Synthesize(constraint, level);
7  return r;
```

Algorithm 4: RepairLoopBody.

```
   Input: P: The buggy program, T: A test suite, s: the
          suspicious statement
   Output: r: A patch for P
1  specification = NULL;
2  tree = NULL;
3  PC[] = NULL;
4  key[] = NULL;
5  value = NULL;
6  foreach t ∈ T do
7      tree = GetSymbolicTree();
8      PC[] = tree.GetPathConstriants();
9      key[] = PC.GetReachableValues();
10     value = PC.value();
11     specification ← ⟨key, value⟩;
12 r = Repair(P, specification);
13 if r! = NULL then
14     return r;
```

(Line 7) by calling the method *RepairLoopConditionAndBody* defined in Algorithm 5. If the suspicious statement is in the loop body, *LoopFix* tries to repair loop body (Line 11) first by method defined in Algorithm 4. If a patch is generated and passes all test cases in the given test suite T (Line 12), the generate patch is returned

Algorithm 5: RepairLoopConditionAndBody.

Input: P: The buggy program, T: A test suite, s: the suspicious statement
Output: r: A patch for P

```
1  specification = NULL;
2  key[] = NULL;
3  foreach t ∈ T do
4  |  key = GetReachableValues();
5  |  if JumpIntoLoopBody(t) == true then
6  |  |  specification ← ⟨key, true⟩;
7  |  else
8  |  |  specification ← ⟨key, false⟩;
9  r = Repair(P, specification);
10 if r! = NULL then
11 |  P′ = ChangeLoopCondition(P, r);
12 |  r = RepairLoopBody(P′, T, s);
```

and *LoopFix* ends properly. Otherwise, we repair loop condition and body (Line 15) by calling the method defined in Algorithm 5. The process terminates if a fix patch is generated (Line 5, 9, 13, 17), or all suspicious statements have been examined.

In the following, we introduce the details of each step in our method.

3.1. Fault localization

The fault localization process aims at identifying the root cause of failure by examining the suspiciousness score of every statement in the program. This process is at statement level. Intuitively, the more frequent one statement occurs in the failing test cases, the more suspicious it is. We apply the Tarantula technique (Jones and Harrold, 2005) in this work, where the suspiciousness of a statement *susp(s)* is defined as follows:

$$susp(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

where *totalfailed* and *totalpassed* represent the number of failing and passing test cases in the test suite, respectively. The *failed(s)* and *passed(s)* represent the number of failing test cases and passing test cases covering statement *s* respectively. If *susp(s)=1*, it indicates that statement *s* exists in all the failing test cases and none of the passing test cases, and thus is considered highly likely to be correlated with the bugs; while *susp(s)=0* indicates that statement *s* never occurs in any of the failing test cases, and thus is less likely to be correlated with the bugs.

Having decided the suspiciousness scores of all the statements, *LoopFix* proceeds to generate fix patch and verify the generated fix patch. Since the patch generation and verification processes are closely correlated, we introduce these two steps together for different types of loop bugs in the following sections.

3.2. Repairing loop condition

3.2.1. Detailed approach

In this section, we present the details of automatic repair of loop condition bugs, with the assumption that there is no bugs in the loop body. Bugs in loop conditions can always be attributed to the incorrect number of iterations. Since we assume that there does not exist any bug in the loop body, for each test case, we can always reach the case when the correct output is obtained by executing the loop repeatedly. During this process, we can obtain a series of ⟨*key, value*⟩ pairs as the *specification*, which can be used as constraints for patch synthesis.

For each test case, we initialize the number of iteration of the loop to 1 and check whether the result is correct. If not, we set it to 2 and check again. We repeat this process until we find the number *N* such that if the loop is executed *N* iterations, the result is correct. And then, we collect the set of runtime ⟨*key, value*⟩ pairs. For each test case, the *key* includes all the reachable variables in the loop condition and the loop body. The *value* is defined as follows. During each iteration, if the return of current iteration equals to the expected output of this test case, the loop is terminated and the *value* is set to *false*. Otherwise, the loop continues executing and the *value* is set to *true* (Line 8 to Line 11). For each test case, the number of ⟨*key, value*⟩ pairs generated is proportional to the number of executed iterations for this test case. The ⟨*key, value*⟩ pairs are added to the *specification* to synthesize fix patch. Formally we have

$$value = \begin{cases} false, & \text{expected output generated} \\ true, & \text{otherwise} \end{cases} \quad (8)$$

Given the *specification* as the input, we can employ the oracle guided component based program synthesis approach (Jha et al., 2010) to synthesize a patch (Line 13). The synthesis technique starts with patch synthesis using level 0 components and incrementally involves higher level components (Line 5) if the synthesis fails to generate a patch with lower level components. This process repeats until components in all levels have been attempted or a patch is successfully synthesized. Finally, we re-execute the whole test suite to check the correctness of the synthesized patch. If the patched program can pass all test cases, the generated patch is considered to be a correct fix.

As shown in Algorithm 3, the variable *specification* is used as one of the inputs to the component based program synthesis approach. We use the variable *level* to represent components in different levels. Initially, the *level* is set to 0. Our algorithm tries to synthesize a patch *r* with the given *constraint* and *level*. If it fails, new components are added and and the same process is repeated until all the components are considered or a patch is successfully synthesized (Line 4 to Line 6).

3.2.2. Example Illustration

Fig. 5 shows the program of an integer reverse problem, which takes an integer and revers it as an output. For example, given an integer 123, the program outputs 321. Table 2 shows a test suite of this program.

Given the program shown in Fig. 5, *LoopFix* first conducts fault localization based on the Tarantula technique. The results show that statement 9 has the highest suspiciousness score. Then, we start from statement 9 (based on the suspiciousness score), and collect the runtime ⟨*key, value*⟩ pairs of this statement. By searching the loop condition, we can get the logical expression of *number* > 9. By searching the loop body, we can obtain two variables, i.e., *temp* and *number*.

In the first test case, the input is 123. In the 1st iteration of executing the loop body, the observed variable value of *number* is 12, *temp* is 3, and the logical expression of *number* > 9 is true. Since the output 3 is not equal to the expected output 321, our algorithm continue to execute the loop and set the value of the ⟨*key, value*⟩ pair to true.

Table 2
The test suite of example 1.

Test case	Input	Expected output	Observed output	Status
1	123	321	32	fail
2	9876	6789	678	fail
3	65815	51856	5185	fail
4	546489	984645	98464	fail

```
1.void func (int number)
2.{
3.    int temp = 0;
4.    if (number == 0)
5.    {
6.        system.out.print(0);
7.        return;
8.    }
9.    while (number > 9)
10.   {
11.       temp = number % 10;
12.       number = number / 10;
13.       system.out.print(temp);
14.   }
15.   return;
16.}
```

Fig. 5. The program with a loop condition bug

Table 3
The ⟨*key, value*⟩ pairs of example 1

Test case 1
⟨{(“number”, 12⟩, ⟨“temp”, 3⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 1⟩, ⟨“temp”, 2⟩, ⟨“number > 9”,false⟩}, true⟩
⟨{(“number”, 0⟩, ⟨“temp”, 1⟩, ⟨“number > 9”,false⟩}, false⟩
Test case 2
⟨{(“number”, 987⟩, ⟨“temp”, 6⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 98⟩, ⟨“temp”, 7⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 9⟩, ⟨“temp”, 8⟩, ⟨“number > 9”,false⟩}, true⟩
⟨{(“number”, 0⟩, ⟨“temp”, 9⟩, ⟨“number > 9”,false⟩}, false⟩
Test case 3
⟨{(“number”, 6581⟩, ⟨“temp”, 5⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 658⟩, ⟨“temp”, 1⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 65⟩, ⟨“temp”, 8⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 6⟩, ⟨“temp”, 5⟩, ⟨“number > 9”,false⟩}, true⟩
⟨{(“number”, 0⟩, ⟨“temp”, 6⟩, ⟨“number > 9”,false⟩}, false⟩
Test case 4
⟨{(“number”, 54648⟩, ⟨“temp”, 9⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 5464⟩, ⟨“temp”, 8⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 546⟩, ⟨“temp”, 4⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 54⟩, ⟨“temp”, 6⟩, ⟨“number > 9”,true⟩}, true⟩
⟨{(“number”, 5⟩, ⟨“temp”, 4⟩, ⟨“number > 9”,false⟩}, true⟩
⟨{(“number”, 0⟩, ⟨“temp”, 5⟩, ⟨“number > 9”,false⟩}, false⟩

In the 2nd iteration, the observed variable value of *number* is 1, *temp* is 2, and the logical expression $number > 9$ is false. Since the output 32 is not equal to 321, the value of the ⟨*key, value*⟩ pair is set to true.

In the 3rd iteration, the observed variable value of *number* is 0, *temp* is 1, and the logical expression $number > 9$ is false. The output is 321, which is equal to the expected output 321. Therefore the loop terminates and the value of the ⟨*key, value*⟩ pair is set to false.

After walking through the first test case, we obtain three ⟨*key, value*⟩ pairs. Similarly we can generate all ⟨*key, value*⟩ pairs for all other test cases, as shown in Table 3. These ⟨*key, value*⟩ pairs are called *specification* (in Algorithm 3) and are used input to the component based program synthesis approach to synthesize a patch for this program.

Suppose we want to replace the loop condition $number > 9$ with a new function $f(!‘)$. There are two runtime variables, i.e., *temp* and *number*. Since the variable *temp* is not initialized, we assume that it cannot be used by $f(!‘)$. Let us assume the loop condition to be $f(number)$. According to the ⟨*key, value*⟩ pairs in the first test case, we know that f must satisfy $f(12) == true$, $f(1) == true$ and $f(0) == false$. We can also obtain constraints that function $f(!‘)$ must satisfy from the other test cases. Program synthesis methods require basic components, such as plus, minus, not, or, to construct $f(!‘)$. In the first trial, only constant is allowed. However, there is no constant function that can satisfy all the three constraints. In the second trial, we allow $f(!‘)$ to use $>$, $<$, $>=$ and $<=$. The function $f(!‘)$ can take the form of $number > c$ (c is a constant). The synthesis approach find a solution $f(number) = number > 0$. The patch for this program is $number > 0$. Finally, we run the verification step to ensure that this patch is correct.

3.3. Repairing loop body

3.3.1. Detailed approach

For the case of repairing bugs that exist only in the loop body, we assume that the loop condition is correct. Then the problem of repairing loop body bugs can be transformed to repairing sequence program bugs by unfolding the loop body. The detailed steps of the repairing algorithm are presented in Algorithm 4.

For each test case, we construct the symbolic tree and the path constraints (PC) of the program (Cadar and Sen, 2011). During this process, we can obtain a series of ⟨*key, value*⟩ pairs from the path constraints as the *specification*, which can be used as constraints for patch synthesis. For each test case, the *key* includes all the reachable variables in the path constraints. The *value* equals to the value of the patch constraints (Line 7 to Line 11).

Given the *specification* as the input, the oracle guided component based program synthesis approach (Jha et al., 2010) is employed to synthesize a patch. Finally we re-execute the whole test suite to check whether all test cases can pass. If all test cases pass, then the generated fix patch for this program is regarded as correct.

3.3.2. Example Illustration

Consider the following scenario. There are four person, A, B, C and D.

A says “I do not get the offer”;
B says “C gets the offer”;
C says “D gets the offer”;
D says “I do not get the offer”.

```
//For the sake of brevity, A=1, B=2, C=3, D=4
 1.int func ()
 2.{
 3.    int k = 0, flag = 0;
 4.    while (k <= 4)
 5.    {
 6.        //fix: flag = (k != 1)+(k == 3)+(k == 4)+(k != 4)
 7.        flag = (k == 1) + (k == 3) + (k == 4) + (k != 4);
 8.        if (flag == 3)
 9.            return k;
10.        else
11.            k ++;
12.      }
13. }
```

Fig. 6. The program with a loop body bug.

Table 4
The ⟨*key, value*⟩ pairs of example 2.

⟨{(“k”, 1), ⟨“(k==1)+(k==3)+(k==4)+(k!=4)”,2)}, !3⟩
⟨{(“k”, 2), ⟨“(k==1)+(k==3)+(k==4)+(k!=4)”,1)}, !3⟩
⟨{(“k”, 3), ⟨“(k==1)+(k==3)+(k==4)+(k!=4)”,2)}, 3⟩

Knowing that only one person tells the truth and only one person gets an offer from the company, the program in Fig. 6 is developed to identify who gets the offer. The program shown in Fig. 6 contains a loop body bug. There is no input and the expected output is 3. But this program has no output.

Based on the Tarantula technique, the statement 7 has the highest suspiciousness score. Targeting at this statement, we collect the runtime ⟨*key, value*⟩ pairs of this program with symbolic execution. Fig. 7 shows the symbolic tree of this program. In the symbolic tree, each node records the variables and the path constraints in the corresponding line.

To simplify the notations, we use $f(k)$ to represent the logical expression $(k == 1) + (k == 3) + (k == 4) + (k!=4)$. By searching the path constraints, we get a variable k and a logical expression $f(k)$. According to the symbolic tree, when the output of the program is 3, the path constraints of symbolic tree are $(f(1)!=3)\wedge(f(2)!=3) \wedge (f(3) == 3)$. Therefore, we can get three ⟨*key, value*⟩ pairs shown in Table 4. These pairs are inputs to the component based program synthesis approach.

Suppose we want to synthesize a new $f(k)$. According to the ⟨*key, value*⟩ pairs, we know that f must satisfy $(f(1)!=3)\wedge(f(2)!=3) \wedge (f(3) == 3)$. Program synthesis requires basic components to construct f. After traveling, we allow f to use $>$, $<$, $>=$, $<=$, + , and - . Finally, we synthesize a fix patch, i.e., $f(k) = (k!=1) + (k == 3) + (k == 4) + (k!=4)$, for this program with our approach. The testing with the given test suite shows that the generated fix patch is correct.

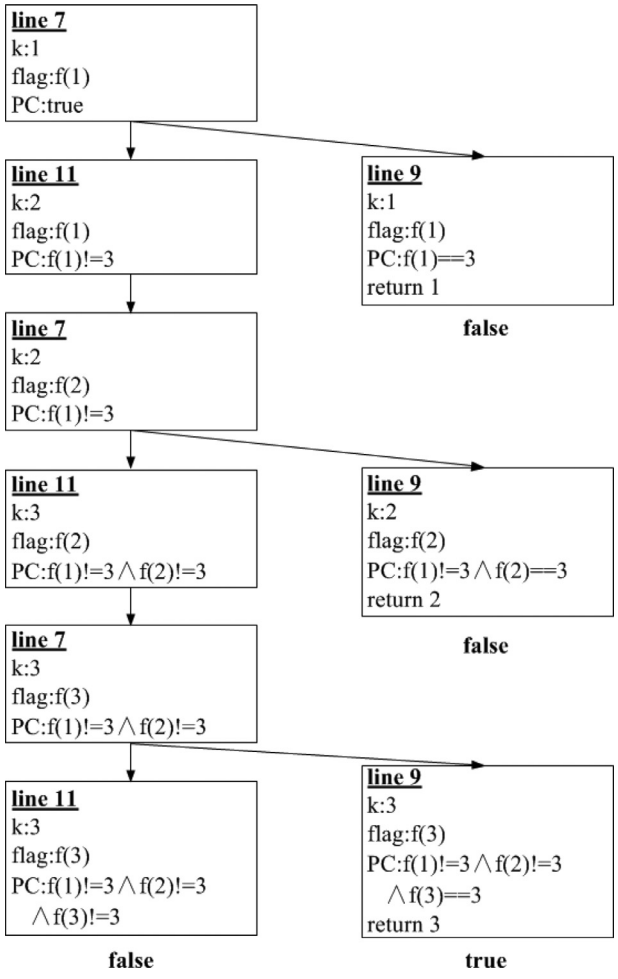

Fig. 7. The symbolic tree of illustrating example 2.

3.4. Repairing loop condition and body

3.4.1. Detailed approach

In this section, we consider the most complex case of repairing bugs i.e., both loop condition and loop body are buggy. Algorithm 5 shows the details of our approach.

Firstly, we synthesize a patch to fix the loop condition bug. In some test cases, the loop condition is satisfied and the loop body is executed. On the contrary, in the other test cases, the loop condition is not satisfied and the loop body can not be executed. It is important to distinguish whether the loop body is executed. And then, we collect the set of runtime ⟨*key, value*⟩ pairs. For each test case, the *key* includes all the reachable variables in the loop condition and the loop body. The *value* is defined as follows. If the loop body is executed, the *value* is set to *true*. Otherwise, the *value* is

```
1.int func (int x)
2.{
3.    int y = 0;
4.    while (x > 0)//fix: while (x > 5)
5.    {
6.        y=y+1;   //fix: y=2*x-10
7.        x--;
8.    }
9.    return y;
10.}
```

Fig. 8. The program with a loop condition bug and a loop body bug.

Table 5
The test suite of example 3.

Test case	Input	Expected output	Observed output	status
1	6	2	6	fail
2	7	4	7	fail
3	8	6	8	fail
4	4	0	4	fail
5	1	0	1	fail
6	5	0	5	fail

Table 6
The former ⟨*key, value*⟩ pairs of example 3.

Test case 1	⟨{(“x”,6), ⟨“x > 0”,true⟩}, true⟩
Test case 2	⟨{(“x”,7), ⟨“x > 0”,true⟩}, true⟩
Test case 3	⟨{(“x”,8), ⟨“x > 0”,true⟩}, false⟩
Test case 4	⟨{(“x”,4), ⟨“x > 0”,true⟩}, false⟩
Test case 5	⟨{(“x”,1), ⟨“x > 0”,true⟩}, false⟩
Test case 6	⟨{(“x”,5), ⟨“x > 0”,true⟩}, false⟩

set to *false* (Line 4 to Line 8).

$$value = \begin{cases} true, & \text{the loop body is executed} \\ false, & \text{otherwise} \end{cases} \quad (9)$$

After the loop condition is synthesized, we replace the original loop condition with the synthesized loop condition (Line 11). In this way, we actually reduce the problem of repairing bugs that exist in both loop conditions and bodies into the problem of repairing bugs in loop bodies. Therefore the same loop body repair techniques introduced in the front is used to generate the fix patch for the loop body (Line 12).

Finally, we apply the fix patch synthesized for both loop condition and loop body and execute the whole test suite. If the patched program can pass all test cases, it indicates that the patch synthesized for this program is correct.

3.4.2. Example Illustration

Fig. 8 shows a simple loop program, which contains bugs that coexist in both the loop condition and body. The test cases for this algorithm are listed in Table 5.

We first compute the suspicious degree of each statement and find that the top two suspicious statements are Line 4 (the loop condition) and Line 6 (the loop body). Then we start from Line 4 to fix the loop condition bug. The ⟨*key, value*⟩ pair we collected for each test case is given in Table 6. For the first three test cases, since their expected output y is different from the initialized value, the loop body must be executed to generate the output and thus their values are defined as true (based on Definition 9). The key consists of two parts, i.e., the variable x and the condition expression $x > 0$. In contrast, for the last three test cases, since their expected output values for y are the same as their initialized values, which indicates the loop body does not need to be executed in order to get the expected value, and thus their values are set to false (based on Definition 9). Using these ⟨*key, value*⟩ pairs as inputs, we can synthesize the fix condition patch $x > 5$.

Table 7
The latter ⟨*key, value*⟩ pairs of example 3.

Test case 1	⟨{(“x”,6), ⟨“x-5”,1⟩, ⟨“y”,1⟩},2⟩
Test case 2	⟨{(“x”,7), ⟨“x-5”,2⟩, ⟨“y”,1⟩},4⟩
Test case 3	⟨{(“x”,8), ⟨“x-5”,3⟩, ⟨“y”,1⟩},6⟩

In the second step, we replace the original loop condition with the synthesized patch $x > 5$ and continue fixing the loop body bug in Line 6. Since after replacing the loop condition, the loop body is not executed for test cases 4 to 6, we only consider the first three test cases, the ⟨*key, value*⟩ pairs of which are shown in Table 7. Given these ⟨*key, value*⟩ pairs as inputs, program synthesis technique for loop body is employed to synthesize the patch $y = 2 * x - 10$. Finally we verify the correctness of the patch by executing all test cases.

4. Evaluation

In this section, we compare the performance of *LoopFix* with Angelix (Mechtaev et al., 2016), S3 (Le et al., 2017b) and JFix (Le et al., 2017a) on real-world projects. We want to answer the following research questions through our evaluations.

1. RQ1: How effective is *LoopFix* in fixing bugs in loop conditions and loop bodies; and
2. RQ2: What is the quality of the fix patches generated by *LoopFix*.

4.1. Experimental settings

We use Symbolic Path Finder (SPF) (Symbolic path finder, 0000) to generate repair constraints. Symbolic Path Finder is a symbolic execution engine which is mainly used for generating high coverage test suites and detecting bugs. The SMT solver used in *LoopFix* is (Z3, 0000). We use default settings of Angelix and S3/JFix. All the experiments are run on a G645 2.90GHz CPU, 4GB memory computer with Ubuntu 10.04.

In the following section, we show the evaluation results of *LoopFix* on three datasets, i.e., IntroClass (Le Goues et al., 2015),

Table 8
Evaluation results on IntroClass.

Project	Buggy Loop	Fixed by LoopFix	Fixed by Angelix	Fixed by S3/JFix
Checksum	2	2, 100.00%	0, 0.00%	1, 50.00%
Digits	2	2, 100.00%	1, 50.00%	1, 50.00%
Total	4	4, 100.00%	1, 25.00%	2, 50.00%

Table 9
Detailed evaluation results on IntroClass.

Type	Buggy code	Patch written by programmers	Patch generated by *LoopFix*
C	$while(cdum == 1)$	$while(cdum!=1)$	$while(cdum > 1 \| \| cdum < 1)$
B	$sum = (accum\%64) + 22;$	$sum = (accum\%64) + 12;$	$sum = (accum\%64) + 12;$
C	$while(number > 9)$	$while(number > 0)$	$while(number > 0)$
B	$sum = character;$	$sum+ = character;$	$sum = sum + character;$

Defects4J (Just et al., 2014) and Loops (Competition on software verification, 2016).

IntroClass (Le Goues et al., 2015) is a set of programs written by UC Davis's students taking an introductory programming course. IntroClass consists of 6 projects: Checksum, Digits, Grade, Median, Smallest and Syllables, out of which only Checksum and Digitals contain 21 and 55 buggy loop statements respectively. The test suites of these two projects include 6 blackbox test cases and 10 whitebox test cases.

Defect4J (Just et al., 2014) is an open source dataset consisting of 5 projects: Chart, Time, Mockito, Lang and Math. It has 262 real bugs, out of which 46 bugs are loop-related. Defects4J is a commonly used benchmark for evaluating bug repair approaches with sufficient test cases.

Loops (Competition on software verification, 2016) is a benchmark used in Competition on Software verification 2016 (SV-COMP 16). It contains 5 loop categories, i.e., loops, loop-new, loop-lit, loop-acceleration and loop-invgen. Since loop-invgen contains many assertions that are not relevant to loops, thus we used the other 4 loop categories, which contains 123 programs (each program contains one loop) in total, in our evaluation.

4.2. Evaluation results

4.2.1. IntroClass dataset

We apply *LoopFix*, Angelix and S3/JFix to automatically repair loop bugs. Table 8 shows the evaluation results on IntroClass. We can see that *LoopFix* can repair all loop bugs. But Angelix can repair one loop bug and S3/JFix can repair two loop bugs. However, even if a patched program can cause all test cases to pass, it does not necessarily indicate that the fix patch is correct. Therefore, we also compare the patches synthesized using *LoopFix* with those patches provided by programmers, and check whether they are functionally equivalent. The last two columns in Table 9 shows that the patches synthesized by *LoopFix* and the patch provided by programmers in these two projects are functionally equivalent.

4.2.2. Defects4J dataset

Table 10 shows the evaluation results on Defects4J. We can see that *LoopFix* can repair around 30% of loop bugs on average. But Angelix can repair 4.35% of loop bugs and S3/JFix can repair 13.04% of loop bugs. Table 11 lists the details of patches synthesized by *LoopFix*, as well as the patches provided by programmers for all the loop bugs in Defects4J. From the results, we observe that *LoopFix* is able to synthesize fix patches that are functionally equivalent to the fixes provided by programmers. In terms of the formats of synthesized fix patches, those synthesized for for loop condition bugs are more similar to programmer provided fix patches than the patches synthesized for loop body bugs. This observation is reasonable since the ways of encoding loop conditions are relatively restricted than that of encoding loop bodies.

4.2.3. Loops dataset

Table 12 shows the evaluation results on Loops. We can see that *LoopFix* can repair more than 30% of loop bugs. But Angelix can repair 3.25% of loop bugs and S3/JFix can repair 13.01% of loop bugs.

4.3. Answering research questions

With the above evaluation results, we try to answer two research questions that are raised in the beginning of this section.

4.3.1. RQ1: how effective is LoopFix in fixing bugs in loop conditions and loop bodies?

As shown in the evaluation results with IntroClass, Defects4J and Loops, *LoopFix* is able to fix 100%, 30.43% and 33.33% of the buggy loops, respectively. *LoopFix* performs more effective than Angelix, S3 and JFix on buggy loops. *LoopFix* is similar to Angelix, S3 and JFix all dealing with loop condition bugs. However, *LoopFix* focuses on all the statements in the loop body, including the if condition, basic assignment statements (including arithmetic/logic expressions) and so on. Angelix, S3 and JFix mainly consider bugs exist in the loop/if conditions. However, there are a great portion of loop bugs which are caused by arithmetic/logic expressions that are not in the loop/if conditions. The buggy statement in the loop body may be executed many times. So it is difficult to fix loop body bugs.

In terms of synthesizing fix patches, *LoopFix* performs better in fixing loop condition bugs than loop body bugs. This is because the statements in loop conditions are usually simple expressions composed of binary or unary operators, e.g., $>$, $<$, and thus are easier for synthesizing techniques to generate patches. However, the type of statements are more diverse and complex in loop bodies, and sometimes it is hard to synthesize patches even the $\langle key, value \rangle$ pairs are collected.

4.3.2. RQ2: what is the quality of the fix patches generated by LoopFix?

For each dataset, we compared the patches synthesized by *LoopFix* with the patches provided by programmers. The results show that the patches synthesized by *LoopFix* are functionally equivalent to the patches provided by programmers. Moreover, in most of situations (especially in patches fixing loop condition bugs), the semantics of those patches are also very similar. We also observe that the patches synthesized by *LoopFix* are more close to natural language semantics, which makes it easy to understand.

Table 10
Evaluation results on Defects4J.

Project	Bug	Buggy Loop	Fixed by LoopFix	Fixed by Angelix	Fixed by S3/JFix
Chart	26	4	1, 25.00%	0, 0.00%	0, 0.00%
Time	27	2	0, 0.00%	0, 0.00%	0, 0.00%
Mockito	38	7	1, 14.29%	0, 0.00%	0, 0.00%
Lang	65	13	4, 30.77%	0, 0.00%	1, 7.69%
Math	106	20	8, 40.00%	2, 10.00%	5, 25.00%
Total	262	46	14, 30.43%	2, 4.35%	6, 13.04%

Table 11
Detailed evaluation results on Defects4J.

Type	Buggy code	Patch written by programmers	Patch generated by *LoopFix*
B	*rowData.removeValue(i);*	*rowData.removeValue(columnKey);*	*rowData.removeValue(columnKey);*
C	*for(Matcherm: uniqueMatcherSet(indexOfVararg))*	*for(int pos = indexOfVararg; pos < matchers.size(); pos++)*	*for(inti = indexOfVararg; i < matchers.size(); i++)*
B	*if (hexDigits > 16 ‖ (hexDigits == 16&&firstSigDigit > 7))*	*if(hexDigits > 16)*	*if(hexDigits > 16)*
B	*classes[i] = (array[i] == null?null : array[i].getClass());*	*classes[i] = array[i].getClass();*	*classes[i] = array[i].getClass();*
B/C	*1.for(int i = startIndex; i <= size; i++)2.if (thisBuf!=ch)*	*1.for(inti = startIndex; i < size; i++) 2.if (thisBuf == ch)*	*1.for(int i = startIndex; i < size; i++) 2.if (thisBuf == ch)*
B	*days+= end.getActualMaxim(Calendar.DAY_OF_MONTH);*	*days+= 31;*	*days = (31 − days)%31*
B/C	*1.while(true) 2.if (iter%n == 0)*	*1.while(true) 2.if(iter%n!=0)*	*1.while(true) 2.if(iter%n > 0‖iter%n < 0)*
B	*res[i] = x[i]/diff;*	*res[i] = (x[i] − boundaries[0][i])/diff;*	*res[i] = x[i]/diff − boundaries[0][i]/diff;*
C	*for(int i = begin; i < begin + length; i++)*	*for(inti = begin; i < begin + weights.length; i++)*	*for(int i = 0; i < weights.length; i++)*
C	*incrementIterationsCounter();*	*++count;*	*count = count + 1*
B	*max = Math.max(max, Math.abs(a));*	*max+= Math.max(max, Math.abs(a));*	*max+= Math.max(max, Math.abs(a));*
C	*for(int i = 0; i < p; ++i)*	*for(int i = 0; i < p − 1; ++i)*	*for(int i = 0; i < p − 1; ++i)*
C	*for(int i = 0; i < 4 ∗ (n − 1); i+= 4)*	*for(int i = 0; i < 4 ∗ n − 1; i+= 4)*	*for(int i = 0; i < 4 ∗ n − 1; i+= 4)*
B/C	*1.for(int i = 4 ∗ i0; i < 4 ∗ n0 − 16; i+= 4) 2.if (end − start > 3)*	*1.for(int i = 4 ∗ i0; i < 4 ∗ n0 − 11; i+= 4)2.if (end − start > 2)*	*1.for(int i = 4 ∗ i0; i < 4 ∗ n0 − 11; i+= 4)2.if (end − start > 2)*

Table 12
Evaluation results on loops.

Project	Buggy Loop	Fixed by LoopFix	Fixed by Angelix	Fixed by S3/JFix
loops	64	20, 31.25%	2, 3.13%	5, 7.81%
loop − new	35	10, 28.57%	1, 2.86%	4, 11.43%
loop − lit	16	6, 37.50%	0, 0.00%	3, 18.75%
loop − acceleration	8	5, 62.50%	1, 12.50%	4, 50.00%
Total	123	41, 33.33%	4, 3.25%	16, 13.01%

Besides the above observations, we also find that there are certain number of circumstances, in which *LoopFix* failed to synthesize fix patches. To summarize, *LoopFix* shows bad performance in the following three situations. (1) We have seen bad performance of *LoopFix* in nested loops. This is because nested loops contain multiple levels of loop conditions, and it is hard for *LoopFix* to pinpoint a certain level of bugs. (2) If there are multiple bugs exist in the loop body, then usually they are correlated and affect each other. Currently *LoopFix* only considers the situation of one bug in the loop body, and thus it is hard to synthesize fix patches for this situation. (3) If the loop body is very complex, e.g., with vectors, lists, class and function calls, and requires context information to understand the bugs, then it is hard for synthesize-based method to generate fixes. Therefore, *LoopFix* does not show satisfying performance in this kind of scenarios.

5. Threats to validity

Independence between buggy loop condition and body. Our approach assumes that loop condition bug and loop body bug are independent. If one variable exists in the buggy loop condition, then it will not lead to a loop body bug. Currently, our approach cannot handle those situations in which the bugs in both the loop condition and the loop body are caused by the same variable.

Limitation of test suites. In our approach, we assume that generated patches are considered correct if they lead the program under repair pass all provided test cases. The patches synthesized by our approach depend on the test suite. The synthesized patches target at passing all test cases in the test suite. Therefore, if we choose another test suite, the synthesized patches may change. This is because our approach is guided by a given test suite, which is a subset of the formal specification. In order to improve the correctness of the synthesized patches, our approach chooses the test suite including as many test cases as possible. But, more test cases lead to overfitting.

Limitation of program synthesis. The quality of the synthesized patches depends on the synthesis algorithm adopted. In this paper, our approach employs the oracle guided component based program synthesis approach to synthesize the patches. This approach is good at synthesizing both mathematical and logic patches. To synthesize high quality patches for other kinds of bugs, different program synthesis approaches can be employed in our approach.

6. Related work

Automatic program repair aims at automatically generate patches to fix software defects and is an active research topic. Existing approaches can be classified as two classes, i.e., search based approaches (Arcuri and Yao, 2008; Le Goues et al., 2012; Goues et al., 2012; Perkins et al., 2009; Kim et al., 2013; Weimer et al., 2013; Qi et al., 2014; Long et al., 2014; Pei et al., 2011; Long and Rinard, 2016; Gao et al., 2015a; Gao et al., 2015b) and constraint solving based approaches (Nguyen et al., 2013; Mechtaev et al., 2015; Mechtaev et al., 2016; DeMarco et al., 2014; Xuan et al., 2017; Le et al., 2017b; Le et al., 2017a; Ke et al., 2015; Long and Rinard, 2015; Tan and Roychoudhury, 2015). Search based approaches are motivated by the idea of search based software engineering. The representative works include GenProg (Le Goues et al., 2012; Goues et al., 2012), PAR (Kim et al., 2013), AE (Weimer et al., 2013), BugSTAT (Zhong and Su, 2015) and RSRepair (Qi et al., 2014). Constraint based approaches first collect repair constraints from the buggy programs, and then transform the patch generation problems into the constraint solving problems. Finally, program synthesis techniques are employed to generate the fix patches. Constraint based approaches are closely related to our work and thus are the focus of this section.

Nguyen et al. (2013) propose a constraint based approach, SemFix, which generates patches for assignments and conditions via semantic analysis. It combines symbolic execution, constraint solving and program synthesis. A constraint is formulated based on the requirements that repaired programs have to pass a given set of test cases. Such a constraint is then solved by iterating over a layered space of repair expressions, layered by the complexity of the repair program. SemFix makes an extension for repairing infinite loops. The core idea is unfolding the infinite loops before symbolic execution. This would lead to a path explosion.

Mechtaev et al. (2015) propose DirectFix, which focuses on the simplicity of patches. DirectFix generates the simplest patch such that the program structure of the buggy program is preserved to the maximal extend. In consideration of simplicity, DirectFix fuses fault localization and repair generation into one step. DirecFix transforms the patch generation problem into the Maximum Satisfiability (MaxSAT) problem, which is an optimization problem. The solution to the MaxSAT problem is converted into the final concise patch. Compared with SemFix, DirectFix is able to generate simpler patches.

Mechtaev et al. (2016) propose Angelix, which is a new constraint based approach. Angelix is more scalable than previously proposed semantics based repair approaches, such as SemFix and DirectFix. Angelix can repair multiple buggy locations that are dependent on each other. Le et al. (2016) assessed different types of syntax-guided program synthesis methods in the core of Angelix and found that the effectiveness of different synthesis engines vary.

DeMarco et al. (2014), Xuan et al. (2017) propose Nopol, which is an approach for automatically repair IF conditions and missing preconditions. Nopol collects data from multiple instrumented test suite executions and transforms it into a SMT solving problem. Then it translates the result from the SMT solver into a source code patch. Nopol is able to repair object-oriented programs and allows the patches to contain nullness checks.

Le et al. (2017b) propose S3, which is a synthesis engine that leverages programming-by-examples methodology to synthesize patches. S3 tackles patches involving multiple lines by grouping multiple buggy locations and synthesizing repairs for several locations at once.

Le et al. (2017a) propose JFix, which is a semantics-based automated program repair framework that targets Java, and an associated Eclipse plugin. JFix is implemented atop Symbolic PathFinder. It extends Angelix, and is designed to be sufficiently generic to support a variety of such techniques.

Ke et al. (2015) propose SearchRepair, which is a repair approach based on semantic code search. SearchRepair encodes human-written code fragments as SMT constraints on input-output behavior. Comparing to GenProg, AE, and RSRepair, SearchRepair is able to solve 20% newly repaired defects.

Long and Rinard (2015) propose staged program repair(SPR), which is a new repair technique using condition synthesis. SPR iterates on multiple trials of pre-defined program transformation schemas one by one, and validates the generated patch with the test suite. In the first stage, SPR chooses one mode to converse the object code. In the second stage, SPR generates abstract conditions by the technique of program synthesis.

Burnim et al. (2009) propose LOOPER, which is an automated technique for dynamically analyzing a running program to prove that it is non-terminating. LOOPER uses symbolic execution to produce simple non-termination arguments for infinite loops dependent on both program values and the shape of heap. The constructed arguments are verified with a SMT solver.

Carbin et al. (2011) propose Jolt, which is a novel system for dynamically detecting and escaping infinite loops. Upon users' request, Jolt attaches to an application to monitor its progress. Based on users' options, Jolt can then transfer control to a statement following the loop, thereby allowing the application to escape the infinite loop and ideally continue its productive execution.

Xiong et al. (2017) study refined ranking techniques for condition synthesis and then proposed ACS system.

The datasets of these approaches consist of real-world bugs. SemFix (Nguyen et al., 2013) and DirectFix (Mechtaev et al., 2015) take the Siemens programs as the dataset. Angelix (Mechtaev et al., 2016) takes the GenProg benchmark as the dataset. S3 (Le et al., 2017b) takes IntroClass and Defect4J as the dataset. JFix (Le et al., 2017a) takes the Commons Math library as the dataset. Nopol (DeMarco et al., 2014; Xuan et al., 2017) takes Defect4J as the dataset.

Different from previous studies on the constraint based approaches, we focus on automatic repair of buggy loop program. *LoopFix* can fix buggy loop condition, loop body and both together.

7. Conclusion and future work

In this paper, we propose an approach *LoopFix* to automatically repair buggy loop programs. By defining the corresponding ⟨*key, value*⟩ pairs for different types of loop bugs, *LoopFix* leverages exiting techniques of symbolic execution and program synthesis to automatically synthesize patches buggy loops. Extensive evaluations on three real-world program sets show that *LoopFix* can effectively repair real bugs in the data sets. To further improve the repair performance, one worthwhile direction is to explore different program synthesis techniques to generate higher quality patches. Another interesting direction is to investigate how to handle the buggy loop structures, which involves inter-dependencies between loop conditions and bodies.

Acknowledgements

This work is partly supported by some projects from National Natural Science Foundation of China, with the project number 61872263,61802275,71502125,61202030. The authors also thank anonymous reviewers for their constructive comments.

References

Arcuri, A., Yao, X., 2008. A novel co-evolutionary approach to automatic software bug fixing. In: Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence). IEEE Congress on. IEEE, pp. 162–168.

Burnim, J., Jalbert, N., Stergiou, C., Sen, K., 2009. Looper: lightweight detection of infinite loops at runtime. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 161–169.
Cadar, C., Sen, K., 2011. Symbolic execution for software testing: three decades later. Commun. ACM 56 (2), 82–90.
Carbin, M., Misailovic, S., Kling, M., Rinard, M.C., 2011. Detecting and escaping infinite loops with jolt. In: European Conference on Object-Oriented Programming. Springer, pp. 609–633.
Competition on software verification 2016. http://sv-comp.sosy-lab.org/2016.
DeMarco, F., Xuan, J., Le Berre, D., Monperrus, M., 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In: Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis. ACM, pp. 30–39.
Gao, Q., Xiong, Y., Mi, Y., Zhang, L., Yang, W., Zhou, Z., Xie, B., Mei, H., 2015. Safe memory-leak fixing for C programs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 459–470.
Gao, Q., Zhang, H., Wang, J., Xiong, Y., Zhang, L., Mei, H., 2015. Fixing recurring crash bugs via analyzing Q&A sites (T). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, pp. 307–318.
Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W., 2012. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In: International Conference on Software Engineering, pp. 3–13.
Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A., 2010. Oracle-guided component-based program synthesis. In: ACM/IEEE International Conference on Software Engineering, pp. 215–224.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault--localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 273–282.
Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, pp. 437–440.
Ke, Y., Stolee, K.T., Le Goues, C., Brun, Y., 2015. Repairing programs with semantic code search (T). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. IEEE, pp. 295–306.
Kim, D., Nam, J., Song, J., Kim, S., 2013. Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 802–811.
Le, X.B.D., Chu, D.H., Lo, D., Goues, C.L., Visser, W., 2017. JFIX: semantics-based repair of java programs via symbolic pathfinder. International Symposium on Software Testing and Analysis.
Le, X.B.D., Chu, D.H., Lo, D., Goues, C.L., Visser, W., 2017. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In: Joint Meeting on Foundations of Software Engineering, pp. 593–604.
Le, X.B.D., Lo, D., Goues, C.L., 2016. Empirical study on synthesis engines for semantics-based program repair. In: International Conference on Software Maintenance and Evolution, pp. 423–427.
Le Goues, C., Forrest, S., Weimer, W., 2013. Current challenges in automatic software repair. Softw. Qual. J. 21 (3), 421–443.
Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W., 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. IEEE Trans. Softw. Eng. 41 (12), 1236–1256.
Le Goues, C., Nguyen, T., Forrest, S., Weimer, W., 2012. GenProg: a generic method for automatic software repair. IEEE Trans. Softw. Eng. 38 (1), 54–72.
Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C., 2006. Have things changed now?: An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability. ACM, pp. 25–33.
Long, F., Rinard, M., 2015. Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp. 166–178.
Long, F., Rinard, M., 2016. Automatic patch generation by learning correct code. In: ACM SIGPLAN Notices, Vol. 51. ACM, pp. 298–312.
Long, F., Sidiroglou-Douskos, S., Rinard, M., 2014. Automatic runtime error repair and containment via recovery shepherding. In: ACM SIGPLAN Notices, Vol. 49. ACM, pp. 227–238.
Mechtaev, S., Yi, J., Roychoudhury, A., 2015. DirectFix: looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 448–458.
Mechtaev, S., Yi, J., Roychoudhury, A., 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In: International Conference on Software Engineering, pp. 691–701.
Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S., 2013. SemFix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 772–781.
Pan, K., Kim, S., Whitehead, E.J., 2009. Toward an understanding of bug fix patterns. Empir. Softw. Eng. 14 (3), 286–315.
Pei, Y., Wei, Y., Furia, C.A., Nordio, M., Meyer, B., 2011. Code-based automated program fixing. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, pp. 392–395.
Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., et al., 2009. Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, pp. 87–102.
Qi, Y., Mao, X., Lei, Y., Dai, Z., Wang, C., 2014. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 254–265.
Symbolic path finder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.
Tan, S.H., Roychoudhury, A., 2015. relifix: Automated repair of software regressions. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, pp. 471–482.
Weimer, W., Fry, Z.P., Forrest, S., 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 356–366.
Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L., 2017. Precise condition synthesis for program repair. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, pp. 416–426.
Xuan, J., Martinez, M., Demarco, F., Clément, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M., 2017. Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. 43 (1), 34–55.
Zhong, H., Su, Z., 2015. An empirical study on real bug fixes. In: IEEE/ACM IEEE International Conference on Software Engineering, pp. 913–923.
Z3. http://z3.codeplex.com.

Wang, Weichao is currently a Ph.D. student of Tianjin University. He received Bachelor degree in computer science and Master degree in software engineering from Peking University in 2011 and 2014 respectively. His research interests include software testing and concurrency program analysis.

Meng, Zhaopeng is currently a Professor in School of Computer Software at Tianjin University and Vice President of Tianjin University of Traditional Chinese Medicine. He received his Ph.D. degree in Computer Science and Technology from Tianjin University. His research interests include Big Data Computing, Internet of Things Software and Systems, Computer Vision and Human-Computer Interaction.

WANG, Zan is currently an associate professor at Tianjin University in China. He received Bachelor degree in applied mathematics, Master degree in computing science and PhD degree in information system from Tianjin University (TJU) in 2000, 2004, and 2009 respectively. He has been a faculty member of Tianjin University since 2010. He was a visiting scholar at UT Dallas in 2013. Zan's research interests include software testing, concurrency program analysis and natural language processing.

LIU, Shuang is currently an associate professor at Tianjin University, China (TJU). She received Bachelor degree in Renmin University of China (RUC) in 2010, PhD degree in computing science from National University of Singapore (NUS) in 2015. She worked as a postdoctoral in SUTD during 2015-2016, and lecturer in SiT during 2016-2017. She has been a faculty member of TJU since October, 2017. Shuang's research interests include software engineering, formal methods and privacy protection.

Hao, Jianye is currently an associate professor in School of Computer Software at Tianjin University. He received his Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong in 2013 and B.E. degree from School of Computer Science and Technology at Harbin Institute of Technology in 2008. His research interests include Artificial intelligence, Multi-agent systems, Machine Learning and Game Theory.