RESEARCH Open Access

Fault localization by analyzing failure propagation with samples in cloud computing environment



Tiantian Wang^{1*}, Kechao Wang² and Xiaohong Su¹

Abstract

With the development of information technology such as cloud computing, IoT, etc, software becomes the infrastructure. On the one hand, it is critical to ensure the reliability of software, on the other, sample code can be mined from open source software to provide reference for automatic debugging. Most of existing automated debugging researches are based on the assumption that defect programs are nearly correct, therefore they can successfully pass some test cases and fail to execute others. However, this assumption sometimes does not hold. In view of the fact that a programs may fail for all the given test cases in the test suite, but there are a large number of examples available for reference, a sample based fault localization method is studied. A fault localization method by analyzing the context of failure propagation is proposed, which locates suspicious statements by identifying the execution status and structural semantics differences between the defective program and sample program. Through the interactive analysis of value sequences and structure semantics, the influence of code variations and failure propagation is reduced. The experimental results have shown that the method can effectively locate suspicious statements and provide assistance for defect repair when there are enough sample programs.

Keywords: Cloud computing, Software safety, Automatic debugging, Fault localization

Introduction

Software becomes the infrastructure of human society. For example, software defines computing, which gives birth to cloud computing. The computing resources can be configured arbitrarily, and the network can be controlled with little management work. All hardware products in the IoT (Internet of Things) era have the characteristics of intelligence and networking. Each product is an intelligent terminal, and the other end of all products must be connected to cloud computing, so these products are all defined by software. All edge devices have broader value because of the definition of software. The intelligent identification, location, tracking, monitoring and management of items can not be separated

from software implementation [1–5]. The development of cloud computing, big data and artificial intelligence requires strong system software support [6–9]. However, bugs often appears in software, which reduce the safety, security, reliability, etc. of the system. Automatic debugging is helpful to improve the efficiency of software development and maintenance as well as the quality of software. Therefore, a lot of researches have been carried out on automatic fault localization [10–12] and repair [13–15] for industrial software. Especially, with the development of open source software, the research of mining open source code to aid automatic debugging has become a research hotspot [16–18].

In the field of computer-aided education, the support of automatic debugging technology is also needed. For massive open online courses with a large number of students, it is difficult for teachers to interact with each student. This brings new challenges to information technology:

Full list of author information is available at the end of the article



 $^{{\}bf *Correspondence: wang tiantian@hit.edu.cn}$

¹School of Computer Science and Technology, Harbin Institute of Technology, XiDazhi Street. 150001 Harbin. China

how to automatically evaluate students' learning effects, provide adequate feedback, and interact with students?. For the practical programming course, the solution of this problem is particularly important. In the process of programming exercises or examinations, a large number of programs implementing the same task can be obtained. Many of these programs are correct and can be used as potential sample programs to guide the fault localization and repair of other defective programs.

This paper focuses on accurately localization of suspicious statements, when sufficient sample programs are provided. The main contributions are as follows. A fault localization method by analyzing the context of failure propagation is proposed, which locates suspicious statements by identifying the execution states and structural semantics differences between the defective program and the sample program. It identifies the equivalent and different statements between the defective program and the sample program, by interactive analysis of execution value sequences and structural semantics. Execution value sequences analysis helps to reduce the impact of code variation on program difference analysis, and structural semantics analysis helps to reduce the impact of failure propagation. These two operations cooperate with each other and help to improve the effectiveness of fault localization. When sufficient sample programs are provided, this method can accurately identify the differences between the defective program and the sample program. Even if the defective programs cannot pass any test case in the given test suite, it can effectively locate the suspicious statements in the defective programs and assist in understanding the causes of failure and repairing defects.

The rest of our paper is organized as follows: the "Related work" section discusses related work of fault localization and automatic debugging. In "Problems of code variation and failure propagation" section, the phenomenon of code variation and failure propagation and their influence on fault localization is analyzed. "Research framework" section presents the overview of the proposed approach. In "Clustering similar programs with code variations" section, the clustering method is described in detail. In "Fault localization by analyzing failure propagation" section, the fault localization method is detailed, whereas "Experimental analysis" section contains the experimental results and evaluation. Finally, conclusions and future work are highlighted in the "Conclusion" section.

Related work

Fault localization

Software fault localization can help developers improve the reliability and safety of information systems, such as cloud computing systems, etc. So many approaches have been proposed. These approaches can be divided into program spectra-based, program slicing, state-based, model-based, program invariant based, mutation analysis, and machine learning-based approach [19–23] etc. These methods have their own advantages and disadvantages [10, 11].

Among them, the program spectra-based method has been widely studied because of its low computational complexity and simple implementation. Program spectra-based methods locate suspicious code by comparing the program spectrum of failure execution and successful execution. Predefined measurement formulas are used to calculate the suspiciousness of each program element (e.g. statement), and the program elements are ranked according to the suspiciousness. The main difference of various program spectra-based methods is that the measurement formulas used are different. Some formulas are equivalent in terms of localization effectiveness [24–26].

Empirical studies have been carried out to evaluate the effectiveness of fault localization tools [27–33], especially the spectra-based methods. Although the conclusion are different from different researches because of different study subjects, some common observations can be got from these researches. To sum up, in order to be well-adopted by practitioners, the following future work of fault localization is needed.

- Fault localization techniques need to be more trustworthy, scalable, able to provide insightful rationales, and integrated to popular IDEs [29].
- In the researches, fault localization and bug fixing are two separate tasks, however, in practice, they are tightly coupled and there is no clear boundary between them [27, 33].
- Browsing the context of failure propagation and providing program comprehension is necessary and helpful in debugging [27, 30].
- Existing work are limited in localizing missing code, however a tool that can recommend additional code to insert maybe helpful in the debugging process [28, 31].
- Existing bug isolation methods are limited in localizing multiple faults, as mis-grouping usually occurs due to the clustering algorithm may not able to divide failed test cases accurately [31, 32].

Besides these observations, it is particularly important to note that except for the program slicing method, most of existing approaches perform fault localization by comparing the execution information of successful test cases and failure test cases [34, 35]. However, defective programs may fail for all the test cases in the given test suite. We call such programs "completely failed programs". In this case, these fault localization methods will not work. Therefore, it is necessary to study the scenarios without successful test cases in the given test suite.

Automatic program repair

It is difficult to obtain a complete formal specification of requirements for industrial software. Therefore, in software fault localization and repair, the execution information of test cases is usually used to assist the analysis. However, only using testing results as the criteria of automatic software repair may cause the overfitting problem. Even if a patched program passes all tests, it may still be incorrect [16, 17]. Therefore other information is needed to complement the test suite. Existing work has shown that example programs mining from open source software, can provide useful information for the generation of patches [18].

Yi et al. studied the performance of current popular industrial software repair tools, GenProg [36], AE [37], Prophet [38] and Angellix [39], in guiding students to debug programs [40]. All the four tools adopt a test-driven approach, that is, if the patch passes all test cases in the test suite, it is considered a successful fix. For example, GenProg first uses program spectra-based method to locate suspicious statements by comparing the statements of successful test cases with those of failed test cases, and assigns suspicious statements a higher probability of selection to be modified. Then genetic programming algorithm is used to transfer correct statements from other locations of the program to be repaired. After continuous evolution, patches that can pass all tests are

searched. The results have shown that the repair rate of these tools is low. One reason is that student programs usually have multiple bugs, which require complex modifications, and often need to introduce new program logic when making modifications. However, the studied tools are difficult to repair missing statements and complex defects with dependencies. Another reason is that these tools use the program spectrabased fault localization method to identify potential modification locations. The program spectra-based method locates suspicious statements by comparing the successful execution and failure execution, while the students' programs may fail for all the given test cases in the test suite. In this case, the spectra-based method can not be applied. The inaccuracy of fault localization results will directly reduce the effectiveness of program repair.

The enlightenment of the above researches to this paper is as follows.

- (1) It is necessary to further study the fault localization method under the scenario of complex defects which lead to the failure of all the given test cases.
- (2) In addition to test cases, sample programs can also be mined from a large number of correct programs to provide auxiliary information for fault localization.

This paper focuses on the sample based fault localization method.

Code A	Code B	Code C				
int fact(int base, int exp){	<pre>int fact(int base, int exp){</pre>	<pre>int fact(int base, int exp){</pre>				
int $i, r=1$;	int <i>j</i> , <i>ret</i> ;	int j, r;				
for(<i>i</i> =0; <i>i</i> < <i>exp</i> ; <i>i</i> ++)	j=0;	j=0;				
r *= base;	ret = 1;	//bug, r uninitialized				
return r;	while $(exp > j)$ {	while $(exp > j)$ {				
}	<i>j=j</i> +1;	<i>j</i> = <i>j</i> +1;				
	ret = ret * base;	r = r * base;				
	return ret;	return r;				
	}	}				
Execution value sequences of fact(3, 4)						
base 3	base 3	base 3				
exp 4	exp 4	exp 4				
i 0,1,2,3	j 0,1,2,3	j 0,1,2,3				
r 3, 9, 27, 81	ret 3, 9, 27, 81	r random numbers				

Fig. 1 Example programs with code variations

Problems of code variation and failure propagation

This paper studies how to locate suspicious statements in defective programs by analyzing the differences of execution state and structure semantics between the defective program and the sample program. However, code variations and failure propagation bring difficulties to program difference analysis and reduce the effectiveness of fault localization.

This section takes the source code in Fig. 1 as an example to analyze the problem. The goal of program codes A, B and C in the figure is to use iterative algorithm to achieve the desired function of computing $base^{exp}$. A and B are correct programs, C is a defective program, lacking the initialization statement of variable r.

(1)Code variation

In addition to the identical representation, there may be two other situations for functionally equivalent programs: one is that they adopt completely different implementations (e.g. different algorithms and data structures); the other is that they adopt the same algorithms and data structures but have different syntactic representations, such as different variable names, control structures, expression forms, statements order, etc. This phenomenon is called code variation [41].

Code variation makes it difficult to identify differences between the defective program and the sample program. For example, in Fig. 1, if the for statement in program A and the while statement in program B can not be identified as equivalent code, when program A is used as the sample program to match with program B, the two loop structures will be identified as difference statements, which will lead to fault location misdetection. In addition, program A and program B have different orders of equivalent statements and variable names. Failure to accurately identify these code variations can also lead to false positives. So, how to recognize code that syntactically different but functionally equivalent, is a key problem to be solved.

There are two advantages of identifying equivalent program with code variations. On the one hand, it can be used to reduce the number of sample programs that need to be provided. For example, only one sample program is needed for program A, program B and program C. On the other hand, it can improve the accuracy of difference analysis between the defective program and the sample program, thereby improving the effectiveness of fault localization.

(2) Failure propagation

The bottom of Fig. 1 shows the sequence of execution values for each variable when fact (3, 4) is called.

Definition (Execution value sequence): Given a test case set T, the sequence of execution values of variable var assigned at assignment statement s_i in program P is as follows:

$$ValueSeqs(T, s_i, var) = \{value(s_i, var, t) | t \in T\}$$
 (1)

 $value(s_i, var, t)$ denotes the values of variable var when executing statement s_i with test case t.

The values of the variable at each assignment statement, got by executing all test cases, constitutes the execution value sequence of the variable. Execution value sequences reflect how the program defines and uses variables, so they imply the execution semantics of the program. Variables that achieve the same function usually have the same value sequence in the execution. These value sequences are not affected by code variations. Especially, the influence of statement order and variable name variation on identifying equivalent codes can be eliminated, by matching execution value sequences. For example, because the variables *j* in program C and *i* in program A have the same value sequence for the same test case, so (j, i) is a matching variable pair. According to the variable pair (j, i), the statements "j = j + 1;" in C and "i++" in A can be matched accurately.

However, unfortunately failure propagation may cause equivalent variables to fail to obtain the same sequence of execution values. For example, due to the lack of initialization statement for variable r in defective program C, the value of r is random number, which is an error state. As the program is executed, the error state is propagated, resulting in r at "r = r*base;" getting a error value sequence. As a result, the statement cannot match the "r*=base;" in program A, but in fact the two statements are equivalent. To sum up, in the case of failure propagation, identifying differences between the defective program and the sample program only by matching the value sequences will also lead to false detection of fault location. Therefore, how to reduce the impact of failure propagation on program difference analysis, is also a key problem.

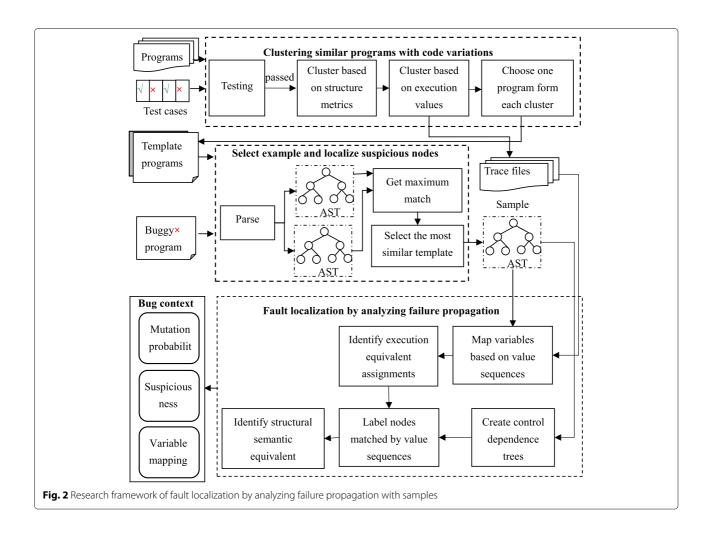
Research framework

Based on the above analysis, a fault localization method by analyzing failure propagation with samples is proposed. The research framework is shown in Fig. 2. The inputs include a defective program, a program set implementing the same task, and a set of test cases. The output is the bug context to provide assistance in understanding and repairing the defects.

(1) In order to identify new reference programs from large scale program sets, a program clustering method based on structural metrics and execution value sequences is proposed. The main steps are as follows.

Step 1: test the programs and collect execution information.

The programs are executed with test cases and the output results are collected. Compare the actual running results of the program with the expected output results of the test cases. If a program pass all the test cases,



it is considered as potentially functionally correct and used as a reference program. Follow-up cluster analysis is performed on such programs.

Step2: perform program clustering based on structural metrics to quickly identify programs with similar structure.

Programs with similar structure are divided into the same cluster, while programs with different structure are divided into different clusters.

Step 3: perform program clustering based on the execution value sequences.

Programs are instrumented based on abstract syntax tree to print the values for each variable. The instrumented programs are executed with test cases, and the execution value sequences are collected. Programs that implement the same algorithm are divided into the same cluster, programs of different algorithms can be divided into different clusters. Furthermore, a template program can be selected from each cluster to form a template set.

(2) Select the sample program to be matched with the defective program.

Computing the similarity of structure metrics and execution value sequences of defective program and template programs, selecting the template program which is most similar to defective program as an sample program for subsequent fault localization. The correct program which is has the highest similarity with the defective program is chosen as the sample program. This is because the more similar, the more likely the sample program adopts the similar algorithm with the defective program.

(3) In order to effectively locate complex defects, reduce the search space of suspicious statements and explain the causes of errors, a fault localization method by analyzing the context of failure propagation is proposed.

This method identifies the difference of execution states and structural semantics [42] between the defective program and the sample program, and locates suspicious statements even in the absence of successful test cases. It focus on solving the problem of code variation and failure propagation.

Firstly, in order to avoid the influence of code variations, especially the statement order variation

and the variable name variation, the execution value sequences of variables in the defective program and the sample program are matched, to identify mapping variables and assignments between these two programs.

Then, in order to reduce the false positives of suspicious statements caused by failure propagation. The structural semantics of the programs is further analyzed based on control dependence trees [42], to identify possible matching variable pairs and statements.

Finally, the different statements identified in the process of structure semantics matching are identified as suspicious statements, and the possible mutation operations to repair the suspicious statements are identified, which helps to analyze the causes of failure, and provides a reference for further repair.

Because this method identifies suspicious statements based on structural semantics difference analysis, it can locate various types of defects, including the missing statements defect. It does not require successful test cases.

The fault localization method can also be integrated into the program repair framework based on genetic programming, which provides support for suspicious statement localization in the absence of successful test cases, and lays a foundation for repairing programs based on samples.

Clustering similar programs with code variations

This paper proposes two levels of program clustering method. Firstly, the method based on structure metrics is performed to quickly identify the programs with similar structures with lower complexity. On this basis, the execution value sequences of variables are further analyzed. By detecting the similarity of the execution sequences, it is possible to recognize functional equivalent programs more accurately, no matter whether the syntactic representations are different. The related definitions are as follows.

Definition (Metric vector): A metric vector is a point in Euclidean space, that is obtained by statistics of the size, structure, and complexity of the abstract syntax tree of the program.

For example, $\nu(\nu 1, \nu 2, \nu 3, \nu 4, \nu 5, \nu 6, \nu 7, \nu 8, \nu 9, \nu 10)$,

v1: the number of nodes; v2: the number of operators; v3: the number of assigned nodes; v4: the number of loop structures; v5: the number of selection structures; v6: the number of system function calls; v7: the number of special data types (such as pointers, arrays and structures); v8: the longest path length; v9: the length of recursive call path; v10: the longest nested path depth of circular.

Definition (Structure similarity) [42]: The structure similarity of two programs is the similarity of their metric vectors as shown in Equation 2.

Structure_sim(
$$v, v'$$
) = 1 - $\sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (v_i - v'_i) / max(v_i, v'_i)^2}$ (2)

In which, ν and ν' are metric vectors.

The structural similarity between programs is computed based on the structural metric vectors statistically calculated based on the abstract syntax trees of the two programs.

Definition (Similarity of two value sequences): Assuming that X_i and Y_j are two execution value sequences, their similarity is defined in Equation 3.

$$Seq_sim(X_i, Y_j) = \frac{LCS(X_i, Y_j)}{(|X_i| + |Y_i|)/2}$$
 (3)

In which, $LCS(X_i, Y_j)$ represents the maximum number of common subsequences in the execution value sequences of X_i and Y_j , where the subsequences may be discontinuous.

Definition (Execution value similarity): Assuming that the execution value sequence set of programs A and B is $S_A = \langle X_1, X_2, X_3, ..., X_i..., X_n \rangle$, $S_B = \langle Y_1, Y_2, Y_3, ..., Y_j..., Y_m \rangle$ respectively, then the execution value similarity of programs A and B is defined in Equation 4.

$$Value_sim(S_A, S_B) = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} MAX(Seq_sim(X_i, Y_j))$$
(4)

The program clustering algorithm is shown in Algorithm 1. The input is a set of programs, and the output is the set of program clusters.

Line 2 to 8 describe the clustering based on structure metrics.

Line 5, in order to analyze the syntax and structural semantic of programs, each program is parsed and its abstract syntax tree is created.

Line 6, the abstract syntax tree is traversed to statistically calculate the metric values, and a metric vector is generated for each program, to approximately evaluate the structure of the program.

Line 8, the programs are clustered and grouped according to the values of *Structure_sim* (see Equation 2) with a hierarchical clustering method.

Line 9 to 20 describe the clustering based on execution value sequences. The result of the structure metrics based clustering is further analyzed.

Line 12, in order to collect the execution information of variables, each program is instrumented based on the abstract syntax tree. Probe statements are inserted into the syntax tree to output the line numbers and variable values of the executed statements.

Algorithm 1 Clustering programs with code variations

Require:

```
TP: a set of template programs, T: A set of test cases
```

Ensure:

21: return Ω

```
\Omega = \omega_1, \omega_2, ..., \omega_k:clustering set
 1: \Omega = \Phi
 2: //clustering based on structure metrics
 3: V = \Phi
 4: for each tp \in TP do
       AST_{tp} = CreateCDT(tp)
       v_{tp} = \text{GetStructureMetrics}(AST_{tp}), V = V \cup \{v_{tp}\}
 7: end for
 8: \Omega_{structure} = HierarchicalCluster(V)
 9: //clustering based on execution value sequences
10: for each omega_i \in \Omega_{structure} do
       for each tp \in omega_i do
          Itp = Instrument(tp)
12:
          tr(Itp) = \Phi
13:
          for each t \in T do
14:
             tr(Itp, t) = \text{Execute}(Itp, t), tr(Itp) = tr(Itp) \cup
15:
          end for
16:
          Seq_{tp} = GetValueSeqs(tr(Itp))
17:
18:
       \Omega_{valuei} = \text{HierarchicalCluster}(Seq_{tp}), \ \Omega = \Omega \cup
19:
20: end for
```

Line 13 to 18, the instrumented program is executed with each test case, to collect the execution value sequences.

Line 19, the programs are clustered and grouped according to the execution value similarity with a hierarchical clustering method. The execution value similarity of two programs, as shown in Equation 3 and 4, is calculated by the longest common subsequence algorithm.

Fault localization by analyzing failure propagation

Execution value sequences analysis is helpful to identify the equivalent codes with code variations in the defective program and the sample program. Therefore, this paper identifies the equivalent variable pairs and equivalent assignment statements in two programs by matching execution value sequences. However, due to the failure propagation, the equivalent assignment statements may not obtain the matched execution value sequence. Therefore, it is necessary to further identify whether the unmatched value sequences is caused by the failure propagation based on the structural semantics, by matching assignment expressions and their context.

Definitions

The combination of dynamic execution value sequences analysis and static structural semantics analysis helps to identify the differences between the defective program and the sample program accurately and locate the suspicious statements in defective programs. The related definitions are as follows.

Definition (Variable pairs matched by value sequences): Given a test case set T, variable var_b at assignment statement s_b in program P_b , variable var_s at assignment statement s_s in program P_s , variable var_b and var_s are variable pairs matched by value sequences, and s_b and s_s are matched assignment statements, if and only if $ValueSeq_s(T, s_b, var_b) \equiv ValueSeq_s(T, s_s, var_s)$, that is, var_b and var_s have the same execution value sequence.

Definition(Control dependence tree): program P's control dependence tree, defined as T = (V, E), is a directed, ordered tree, where V is a set of nodes, representing statements and predicates in the program, E is a set of control dependency edges and procedure call edges [39]. The following attributes are satisfied.

- (1) T is a directed tree. The control dependence edge from node u to node v represents the control dependence, and v control depends on u. This means that whether node v executes or not depends on the value of predicate in node u.
- (2) T is an ordered tree. If node v and node u have the same parent node, they are called brothers. If v is the left brother of u, v precedes u in the execution.
- (3) The node is a triple (*id*, *type*, *exp*), where *id* is the line number of the statement in the program, *type* is the statement type, *exp* is the abstract syntax tree of the expression.
- (4) The procedure call edge represents the call relationship between functions, and connects the function call node and the entry node of the function being called.

Based on the control dependence tree to perform structural semantic analysis, the variable pairs matched by structural semantics are further identified.

Definition(Variable pairs matched by structural semantics): Given a test case set T, variable var_b at assignment statement s_b in program P_b , variable var_s at assignment statement s_s in program P_s , variable var_b and var_s are variable pairs matched by structural semantics, and s_b and s_s are matching assignment statements, if and only if in the control dependence tree of P_b and P_s , the syntax trees of s_b and s_s are completely matched, and both the control dependent paths of s_b and s_s have the same type of nodes.

Fault localization algorithm

Fault localization by analyzing the context of failure propagation is shown in Algorithm 1. The inputs include a defective program, the sample program, and a test case set. The goal is to identify the statements that are different

between the defective program and the sample program as suspicious locations, return mutation operations that may be performed to repair defects, and map variables in the defective program and the sample program to assist program repair.

Lines 1 to 5, instrument and execute the defective program and the sample program with each test case, so as to collect the execution value sequence of each assignment variable.

Lines 6 to 9, map the equivalent variables and assignment statements in the defective program and the sample program by matching the execution value sequences.

Line 10, create the control dependence tree for the defective program and the sample program respectively.

Lines 11 to 15, and Lines 16 to 20, mark the equivalent assignment statements matched by the execution sequences in the two control dependence trees, respectively.

Line 21, The dynamic programming algorithm is used to find the maximum matching between two control dependence trees. On the one hand, the equivalent assignments and the variable pairs matched by structural semantics are identified to reduce the false positives of suspicious statements caused by failure propagation. The variable pairs are added to the variable mapping table. On the other hand, the different statements in the two programs are assigned higher suspiciousness, the mutation operations (insertion, deletion, replacement) that may repair the suspicious statements are identified and given higher mutation operation probability.

Finally, The variable mapping table, mutation operation probability, suspicious location and probability are output to assist understanding the causes of errors. Statement nodes with high probability are suspicious, and mutation operation with high probability is a more likely editing operation to repair the suspicious statement. The control dependence trees with the matching marks can also be output to assist the understanding of the failure context.

A sample

Figure 3 is an example of the structural semantic analysis of the programs in Fig. 1. By matching the subtrees of the program control dependence trees, the similar nodes and different nodes of the defect program C and the sample program A are identified, so as to localize the nodes that may contain defects, identify the possible mutation operations to repair the defects, and narrow the search scope.

Each node in the node sequence of the defect program and the sample program is given a weight, the mismatched node is given a higher probability (value is 0.8), and the matched node is given a lower probability (value is 0.2). The unmatched nodes in the defective program are regarded as suspicious statement. The

Algorithm 2 Fault localization by analyzing failure propagation

```
Require: P_b: A buggy program,
```

```
P_s: A sample program,
    T: A set of test cases;
Ensure:
    SS:SuspiciousStatements,
    Mu:Mutations,
    Var:MapVaraibles;
 1: IP_b = Instrument(P_b), IP_s = Instrument(P_s)
 2: for each item t \in T do
       tr(IP_b, t) = Execute(IP_b, t)
       tr(IP_s, t) = Execute(IP_s, t)
 5: end for
 6: SeqP_b = GetValueSeqs(tr(IP_b))
 7: SeqP_s = GetValueSeqs(tr(IP_s))
 8: MapVariables(SeqP<sub>b</sub>, SeqP<sub>s</sub>)
 9: Map = MapAssignement(P_b, P_s)
10: CDT_b = CreateCDT(Pb), CDT_s = CreateCDT(P_s)
11: for each node v_i \in CDT_b do
      if v_i.pos \in Map then
         mark v_i
       end if
15: end for
16: for each node v_i \in CDT_s do
       if (v_i.pos, v_j.pos) \in Map then
17:
18:
         mark vi
```

unmatched nodes in the sample program are likely to be statement nodes that can be used to repair defects.

Experimental analysis

Research questions

end if

22: **return** SS, Mu, Var

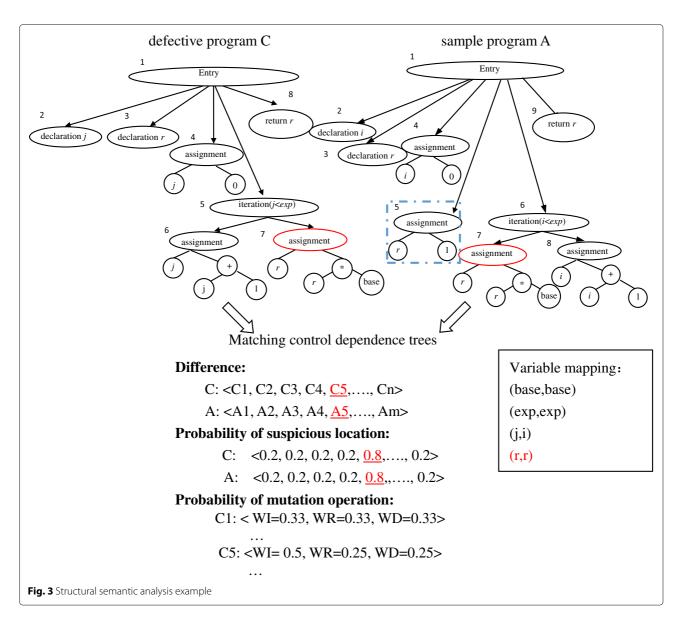
20: **end for**

19:

The research questions are as follows.

21: SS, Mu, Var=MaxMatch(CDT_b , CDT_s)

- (1) RQ1: What is the effectiveness of the program clustering method? Ideally, programs with similar structure and semantics but possibly with code variations should be divided into the same cluster, and programs with different implementation logic or algorithms should be divided into different clusters.
- (2) RQ2: What causes a program failed on all the test cases?
- (3) RQ3: Can the proposed fault localization method effectively identify the differences between the defective programs and the sample program? Which factors affect the effectiveness of fault localization?



Benchmarks

Yi et al. provided a benchmark, as shown in Table 1, when studying the performance of industrial software repair tools in debugging students' programs [40].

These programs are actually written by students in the course of introductory programming. Each experiment has 4-8 programming tasks. The principle of Yi et al. in choosing programming tasks is to cover as many syntactic structures and algorithms as possible. The correct versions and the buggy versions are distinguished by comparing the actual and expected results. In the benchmark, each programming task provides a correct template program and multiple program pairs (*Pb*, *Pc*). *Pb* is a defective program version, which failed on one or more test cases. *Pc* is a modified version of *Pb*, which is obtained by the same person who modifies *Pb*. *Pc* can pass all test cases

Table 1 Benchmark of programs to be clustered and localized

	Table 1 Benefittank of programs to be clastered and localized						
Labs TasksBuggy VersionsCorrect VersionsDescription							
Lab3 4	63	67	Expressions, printf, scanf				
Lab4 8	117	125	Conditionals				
Lab5 8	82	90	Loops, Nested Loops				
Lab6 8	79	87	Integer Arrays				
Lab7 8	71	79	Character Arrays and Functions				
Lab8 6	33	39	Multi dimensional Arrays				
Lab9 8	48	56	Recursion				
Lab108	53	61	Pointers				
Lab118	55	63	Algorithms				
Lab128	60	68	Structures				

in the test suite. There are 661 defective programs in the data set, all of which contain logic errors and no syntactic errors.

There are two reasons for our experimenting on this benchmark. The first one is that it provides not only the buggy versions, but also the correct versions which helps in verify our sample based approach. The second one is that we want to test the capability of our method in deal with programs with complex bugs that cannot pass any test case, this benchmark provides such buggy versions.

Clustering effect analysis

To answer RQ1, the correct versions of each programming task were clustered to select template programs. Ideally, the programs in each cluster are structurally and semantically equivalent, and the programs in different clusters should adopt different implementation logic or algorithms. In order to improve the efficiency of code review, Gumtree [43] was used to generate syntax tree differences between two programs, and then these differences were manually examined to determine whether the two programs should belong to the same cluster.

Clustering result is related to distance threshold. The smaller the distance threshold is, the higher the similarity between programs in the same cluster will be, and the more clusters will be generated. It is expected that the programs in the same cluster should be as similar as possible to avoid filtering out too many template programs to provide sufficient template programs, so the distance threshold is set to 0.1.

The clustering effectiveness was evaluated by *Purity* and *Entropy*. *Purity* calculates the proportion of the correct clustered programs to the total number of programs. *Entropy* reflects the uncertainty of clustering results.

The average *Purity* is 0.95576, and the average *Entropy* is 0.15497. The clustering partition and the actual expectation partition are in good agreement. Therefore, the program clustering method is effective.

Cause analysis of complete failure

To answer RQ2, the benchmark provided by Yi et al. are further analyzed [40]. Among 661 defective programs, 206 defective programs failed to pass any test case in the test suite. Because there is not any successful test case information, these programs cannot analyzed by traditional fault localization methods, such as spectra-based method. This subsection focuses on the analysis of these programs, which are called "completely failed programs".

In the complete failed program set, 86 programs only contain one defect and 120 programs contain multiple defects. Figure 4 counts the number of program versions containing single defect and multiple defects in each programming task.

As shown in Table 2, the types of defects in completely failed programs are various. For example, an operator error, a logical expression error, or an output format error may cause the programs failed on the whole test case set, and the interaction of different kinds of defects may also lead to completely failure. In the completely failed program set, the single defective program mainly contains the following types of defects: input and output format, logical expression, arithmetic expression, variable initialization, variable assignment, array subscript, function parameter errors, etc. The defect types of multiple defect programs include not only the defect types in the single defective programs mentioned above, but also the large concepts missing, complex control structure errors, function invocation errors, statement order errors and so on.

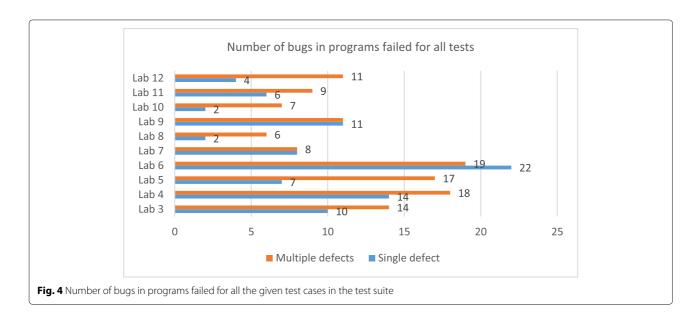


Table 2 Types of bugs which cause programs failed on all the given test cases in the test suite

ID	Defect type	Number	Description	
1	output error	118	output format error, missing or redundant output statements	
2	input error	21	input format error, missing or redundant input statements	
3	logical expression error	36	error logical expressions	
4	variable-related errors	30	variable initialization missing or errors in assignments, missing variables and corresponding processing statements	
5	larger conceptual deficiencies	28	missing key statements or conceptual errors	
6	control structure error	26	loop or branch decision control flow error or missing	
7	function error	22	missing function calls or function definitions, errors in parameters and arguments	
8	array subscript error	17	increase or decrease array access expressions, array sizes	
9	statement order error	10	errors in statement order lead to errors in calculation results	
10	arithmetic expression error	7	incorrect arithmetic expression	
11	other incorrect uses	4	add semicolons before for loop body, string terminator errors, struct $->$ and. misuse	

A program may also contain many types of defects. These defects interact with each other, resulting in the failure of the program on the entire test case set.

Effectiveness analysis of fault localization

To answer RQ3, the effectiveness of fault localization on defective programs failed on all test cases in three cases was analyzed.

Case1: Provide sufficient sample programs. Using the original data in the benchmark, that is, there is a corrected version of P_c corresponding to each defective program P_b .

Case 2: Perform clustering to select sample program. A program is randomly selected from each cluster and added to the template set. In this case, for a defective program P_b , in the process of fault localization, the matching sample program may not be its initial corresponding version P_c , but semantically equivalent to the P_c , but there may be multiple correct versions such as expressions, variable names, statement order, etc.

Case 3: The sample program is inadequate. On the basis of Case2, the set of template programs is reduced. From the set of template programs generated by clustering algorithm, 1/2 and 1/3 number of template programs are randomly selected to form a new template set.

Table 3 counts the number of false positives of fault locations in three cases. It is considered that the defects in the program are correctly localized only when the location of defects is identified as suspicious. If a correct statement is identified as suspicious statement, it is considered as a false positive.

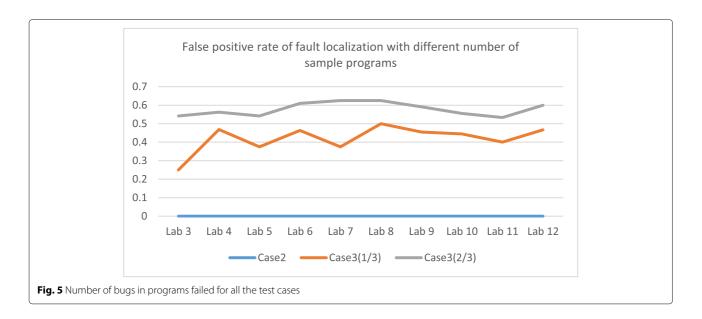
The following conclusions can be drawn from Table 3 and Fig. 5.

 In the case of providing the corresponding correct program as the sample program, this method can

- accurately identify these defect locations as difference statements, and the corresponding correct statements in the example program as potential repair statements. Moreover, the effectiveness of fault localization is not limited by the number and type of defects.
- When the sample program is selected by program clustering, the defect statement can still be accurately localized, which shows that the proposed clustering method combined with fault localization method can effectively localize various defects, including the missing code defect.
- When the number of templates is small and the sample program provided is insufficient, the correct statement will be misreported as suspicious statement. The fewer sample programs available, the greater the probability of false positives. Therefore, the precondition for the effectiveness of this method is that there are enough sample programs available.

Table 3 False positives of fault localization in three cases

Labs	Versions	Case1	Case2	Case3(1/3)	Case3(2/3)
Lab 3	24	0	0	6	13
Lab 4	32	0	0	15	18
Lab 5	24	0	0	9	13
Lab 6	41	0	0	19	25
Lab 7	16	0	0	6	10
Lab 8	8	0	0	4	5
Lab 9	22	0	0	10	13
Lab 10	9	0	0	4	5
Lab 11	15	0	0	6	8
Lab 12	15	0	0	7	9
Tatol	206	0	0	86	119



Conclusion

A fault localization method based on sample programs by analyzing failure propagation is proposed. The experimental results have shown that, when sufficient sample programs are provided, even if the defective program can not pass any test cases, the proposed methods of program clustering and fault localization working together can effectively locate the suspicious statements.

With the development of swarm intelligence and open source software, this method is expected to be applied to industrial software and learn to debugging from open source code, so as to improve the software reliability for information systems such as cloud computing and IoT systems. We will collect example in open source code and verify the effectiveness of our approach in industrial systems in the future work.

Acknowledgements

There is no acknowledgement.

Authors' contributions

TW, and KW are the main researcher, XS are coauthor. All authors read and approved the final manuscript.

Authors' information

Tiantian Wang is an Associate Professor in School of Computer Sceicence and Technology, Harbin Institute of Technology. She received the Doctor degree in Computer Science from Harbin Institute of Technology, China, in 2009. Kechao Wang is an Associate Professor in School of Information Engineering, Harbin University. He received the Doctor degree in Computer Science from Harbin Institute of Technology, China, in 2019.

Su XiaoHong, is a Professor in School of Computer Sceicence and Technology, Harbin Institute of Technology. She received the Doctor degree in Computer Science from Harbin Institute of Technology, China, in 2003.

Funding

This work was supported by the National Natural Science Foundation of China(Grant No. 61977020), Natural Science Foundation of Hei Longjiang Province (Grant No. LH2019F046), and Harbin science and technology innovation talents research project(Grant No. 2016RAQXJ013).

Availability of data and materials

There is no supporting data available.

Competing interests

The authors declare that they have no competing interests.

Author details

¹ School of Computer Science and Technology, Harbin Institute of Technology, XiDazhi Street, 150001 Harbin, China. ² School of Information Engineering, Harbin University, ZhongXing Street, 150086 Harbin, China.

Received: 25 December 2019 Accepted: 3 March 2020 Published online: 12 March 2020

References

- X. Liu, R. Zhu, B. Jalaian, et al., Dynamic spectrum access algorithm based on game theory in cognitive radio networks. Mob Netw Appl. 20(6), 817–827 (2015)
- R. Zhu, X. Zhang, X. Liu, et al., ERDT: Energy-efficient Reliable Decision Transmission for Cooperative Spectrum Sensing in Industrial IoT. IEEE Access. 3, 2366–2378 (2015)
- 3. X. Liu, R. Zhu, A. Anjum, et al., Intelligent data fusion algorithm based on hybrid delay-aware adaptive clustering in wireless sensor networks. Futur Gener Comput Syst. **104**, 1–14 (2020)
- Z. Chen, H. Cai, Y. Zhang, et al., A novel sparse representation model for pedestrian abnormal trajectory understanding. Expert Syst Appl. 138, 112753 (2019). https://doi.org/10.1016/j.eswa.2019.06.041
- Z. Huang, X. Xu, J. Ni, et al., Multimodal representation learning for recommendation in internet of things. IEEE Internet Things J. 6(6), 10675–10685 (2019)
- B. Wu, T.T. Cheng, T.L. Yip, et al., Fuzzy logic based dynamic decision-making system for intelligent navigation strategy within inland traffic separation schemes. Ocean Eng. 197, 106909 (2020)
- W. Wei, X. Fan, H. Song, et al., Imperfect information dynamic stackelberg game based resource allocation using hidden Markov for cloud computing. IEEE Trans Serv Comput. 11(1), 78–89 (2016)
- Z. Huang, X. Xu, H. Zhu, et al., An efficient group recommendation model with multiattention-based neural networks. IEEE Trans Neural Netw Learn Syst (2020). https://doi.org/10.1109/TNNLS.2019.2955567
- L. Dong, Q. Guo, W. Wu, Speech corpora subset selection based on time-continuous utterances features. J Comb Optim. 37(4), 1237–1248 (2019)
- W.E. Wong, R. Gao, Y. Li, et al., A Survey on software fault localization. IEEE Trans Softw Eng. 42(8), 707–740 (2016)

- K.C. Wang, T. Wang, X. Su, P. Ma, Key scientific issues and state-art of automatic software fault localization, 2015. Chin J Comput. 11, 2262–2278 (2015)
- D. Zou, J. Liang, Y. Xiong, et al., An Empirical Study of Fault Localization Families and Their Combinations. IEEE Trans Softw Eng. https://doi.org/ 10.1109/TSE.2019.2892102
- L. Gazzola, D. Micucci, L. Mariani, et al., Automatic software repair: a survey. IEEE Trans Softw Eng. 45(1), 34–67 (2019)
- 14. J. Xuan, Z. Ren, Z. Wang, X. Xie, H. Jiang, Progress on approaches to automatic program repair. J Softw. **27**(4), 771–784 (2016)
- A. Ghanbari, S. Benton, L. Zhang, in Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2019. Practical program repair via bytecode mutation (ACM Press, 2019). https://doi.org/10.1145/3293882.3330559
- J. Liang, Y. Hou, S. Zhou, J. Chen, Y. Xiong, G. Huang, in 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). How to explain a patch: an empirical study of patch explanations in open source projects (IEEE, 2019). https://doi.org/10.1109/issre.2019.00016
- Y. Xiong, X. Liu, M. Zeng, et al., in Proceedings of the 40th International Conference on Software Engineering - ICSE '18. Identifying patch correctness in test-based program repair (IEEE, Piscataway, 2018), pp. 789–799
- J. Jiang, Y. Xiong, H. Zhang, Q. Gao, et al., in *International Symposium on Software Testing and Analysis*. Shaping program repair space with existing patches and similar code (ACM, New York, 2018), pp. 298–309
- W.E. Wong, Y. Qi, BP neural network based effective fault localization. Int J Softw Eng Knowl Eng. 19(04), 573–597 (2009)
- A. Dutta, R. Sahay, P. Mitra, et al., in 2019 IEEE Region 10 Conference (TENCON). Predicate Proximity in Failure: An MLP based Fault Localization approach (IEEE, Piscataway, 2019), pp. 936–941
- 21. X. Li, W. Li, Y. Zhang, et al., in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization (ACM, New York, 2019), pp. 169–180
- Z. Zhang, Y. Lei, X. Mao, et al., in IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks (IEEE, Piscataway, 2019), pp. 445–455
- J. Xuan, M. Monperrus, in 2014 IEEE International Conference on Software Maintenance and Evolution. Learning to Combine Multiple Ranking Metrics for Fault Localization (IEEE, 2014). https://doi.org/10.1109/icsme.2014.41
- L. Naish, H.J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis. ACM Trans Softw Eng Methodol. 20(3), 11 (2011)
- X. Xie, T.Y. Chen, F.C. Kuo, et al., A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Methodol. 22(4), 31 (2013)
- C.M. Tang, W.K. Chan, Y.T. Yu, et al., Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans Reliab. 66(2), 403–424 (2017)
- X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, B. Xu, in *Proceedings of the 38th International Conference on Software Engineering ICSE '16*. Revisit of automatic debugging via human focus-tracking analysis (ACM Press, 2016). https://doi.org/10.1145/2884781.2884834
- X. Xia, L. Bao, D. Lo, et al., in IEEE International Conference on Software Maintenance and Evolution. Automated debugging considered harmful considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems (IEEE, Piscataway, 2017), pp. 267–278
- P.S. Kochhar, X. Xia, D. Lo, et al., in *International Symposium on Software Testing and Analysis*. Practitioners' expectations on automated fault localization (ACM, New York, 2016)
- C. Parnin, A. Orso, in Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11. Are automated debugging techniques actually helping programmers? (ACM Press, 2011). https://doi. org/10.1145/2001420.2001445
- 31. P. Pearson, J. Campos, R. Just, et al., in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). Evaluating and improving fault localization (IEEE. 2017), https://doi.org/10.1109/icse.2017.62
- Z. Li, Y. Wu, Y. Liu, An empirical study of bug isolation on the effectiveness of multiple fault localization (IEEE, 2019). https://doi.org/10.1109/qrs.2019. 00016
- K. Liu, A. Koyuncu, T.F. Bissyandé, et al., You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems (IEEE, 2019). https://doi.org/10.1109/icst.2019. 00020

- 34. R. Gao, W.E. Wong, MSeer—An Advanced Technique for Locating Multiple Bugs in Parallel. IEEE Trans Softw Eng. **45**(3), 301–318 (2019)
- M. Zhang, Y. Li, X. Li, et al., An empirical study of boosting spectrum-based fault localization via PageRank. IEEE Trans Softw Eng. https://doi.org/10. 1109/TSE.2019.2911283
- C.L. Goues, T.H. Nguyen, S. Forrest, et al., GenProg: a generic method for automatic software repair. IEEE Trans Softw Eng. 38(1), 54–72 (2012)
- W. Weimer, Z.P. Fry, S. Forrest, et al., in *Interntional conference on Automated Software Engineering*. Leveraging program equivalence for adaptive program repair: models and first results (IEEE, Piscataway, 2013), pp. 356–366
- 38. F. Long, M.C. Rinard, Automatic patch generation by learning correct code. Symp Princ Programming Lang. **51**(1), 298–312 (2016)
- 39. S. Mechtaev, J. Yi, Roychoudhury A, Angelix: scalable multiline program patch synthesis via symbolic analysis (ACM, New York, 2016), pp. 691–701
- J. Yi, U.Z. Ahmed, A. Karkare, S.H. Tan, A. Roychoudhury, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering ESEC/FSE 2017*. A feasibility study of using automated program repair for introductory programming assignments (ACM Press, 2017). https://doi.org/10.1145/3106237.3106262
- 41. T. Wang, X. Su, Y. Wang, et al., Semantic similarity-based grading of student programs. Inf Softw Technol. **49**(2), 99–107 (2007)
- T. Wang, K. Wang, X. Su, et al., Detection of semantically similar code. Front Comput Sci. 8(6), 996–1011 (2014)
- 43. J.R. Falleri, F. Morandat, X. Blanc, et al., in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering ASE '14.*Fine-grained and accurate source code differencing (ACM Press, 2014). https://doi.org/10.1145/2642937.2642982

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ▶ Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com