Fault Localization Method of Software Defects based on Dependencies Analysis of Program Structure

Hui He[1,2], Lei Zhao[1], Qiao Li[1], Weizhe Zhang[1], Dongmin Gao[1] and Yongtan Liu[2]
[1]*School of Computer Science and Technology,*
[2]*School of Electronics and Information Engineering,*
Harbin Institute of Technology, Harbin, HL, China
{hehui,wzzhang}@hit.edu.cn

Abstract

Software defects are the major risks of system stable operation. Its error localization technology of automation is one of the key research content for trust computing and software assurance. In this paper, we have proposed a new model, which integrates the current methods by analyzing the program structure. We put forward a new automated fault localization method, this method without the degree loss of automation, at the same time, particle size down to basic positioning code statements, makes the position more accurately. We have designed an experiment to examine the effectiveness of CPD base on SIR. And we proved that the CPD is more effective.

Keywords: *Software assurance, Fault localization, Data dependent, Coverage rate*

1. Introduction

Software testing is an important means of defect detection. And it is also not replaced by defect detection methods within the rest stages in the software development process [1, 2]. This is because in the development phases before the software testing, the software itself is the only object of defect detection considering, including source code, requirements analysis and design documents. And only in the testing phase, the software has a run-time environment. At the same time, a lot of defects will be triggered only in a specific runtime environment and lead to software failure. Therefore, software analysis and software testing play a pivotal role to improve software quality and reliability.

Software fault localization and diagnosis is a critical step in the software testing. Only correctly diagnose software failure and locate code with defects in the software can be modified from the sources. Therefore, we have carried out a study of their advantages and defects, and put forward a new mode, named Reverse Data Dependence Analysis Model, which integrates the two methods by analyzing the program structure. On this basis, we finally put forward a new automated fault localization method, which is named CPD. This method is not only the automation lossless, but also changes the basic location unit into single sentence, which makes the location effect more accurate.

2. Reverse Data Dependence Model based on Analysis of Program Structure

In current fault localization methods, the most common methods are CBFL and program slicing. Here we will discuss these two methods, analyze defects of them in fault localization.

2.1 Defects of CBFL and program slicing fault localization

CBFL methods compute defect tendency value of code segment in isolation. The difference between the various methods is reflected on the computational formulas of debugging tendency value, but core ideas of them are the same [3, 4]. However, during program execution, the code segment is not isolated but interdependent. Especially after false data generated, the continuous code segments will be affected, resulting in a successive erroneous data. Therefore, it's likely to produce error in results while calculating the defect tendency in isolation. CBFL method is equivalent to calculating the failure to perform the code segment covered defect tendency. And this equivalent calculation method simplifies the process of program execution, thus affects the accuracy of location.

Program slicing technique is based on the different functions of codes. It picks up codes implementing same function from the source and assembles them into a new code segment [5]. Program slicing technology plays an important role on understanding the program function, and there are some achievements applied to fault localization in the software testing phase. However, the growth of software scale has led to the expansion of the code space. It causes, even with the traditional program slicing technology, the problem of huge amount of sliced codes, and also makes the fault localization become difficult by using program slicing technology [6, 7].

2.2 Reverse data dependence analysis model

Based on the above description of the problem, we propose the reversing data dependent model. Reversing data dependency analysis mainly simplifies the dependencies in program execution paths, aiming to reduce the amount of sliced codes.

Reverse data dependence analysis focuses on the same program execution path to analyze the data, extract and store the data dependencies on this path. Then traverse the stored data dependence reversed, based on a particular variable. Find code statements related to the specific variable, and achieve the purpose of streamlining program execution path, to facilitate the subsequent calculation of code coverage information. Figure 1 shows the execution flow of reverse data dependence analysis.

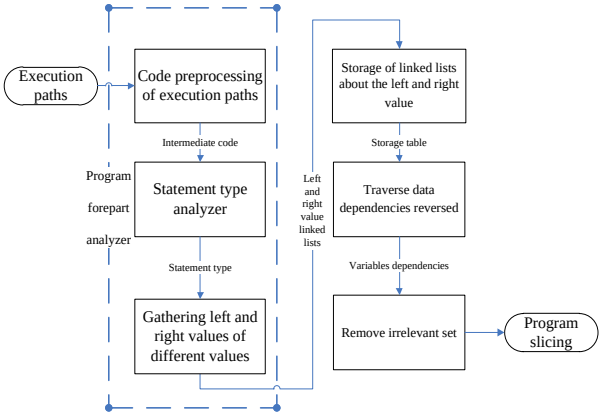


Figure 1. Data dependence extraction function data flow diagram

Preprocessing: The code structure of C language is flexible, and different programming habits of everyone result in different code structures. Without preprocessing before analyzing

code, a great deal of difficulty will appear next. So preprocessing converts various style codes into a unified specification code.

Statement type Analyzer: Because different types of statements need different extraction ideas, it is necessary to analyze the type of statement. Confirm types of statements analyzed by lexical analysis and syntax analysis, such as the basic assignment statement or control statement.

Gather the left value and the right value: Different extraction methods are designed for different specific statements, to collect left values and right values of different statements.

Storage of linked lists about the left and right value: Confirm the linked lists storage structure for left and right values, for the subsequent seeking variables by traversing as a matter of convenience. Good storage structure will improve the efficiency of searching.

Traverse data dependencies reversed: Taking the reverse traversal, traverse the data dependencies table from the end.

Remove irrelevant set: Remove irrelevant statement set with specified variables, and the remaining relevant statement set shall be the final program slicing.

3. CPD Fault Localization Model

Considering the defects and advantages of both CBFL and program slicing fault localization, and combining with the theoretical basis of the analysis of program structure, we propose a fault localization solution CPD (Coverage and Program Data slicing), based on program execution path data dependency analysis.

N test cases of a complete program correspond to N different execution paths. Combining all test cases with the control flow analysis, a number of different execution paths can be got. These execution paths contain different code statements, and the data stream is passed between these code statements. That is, when a variable is defined or assigned, its data dependencies are passed on as the program's execution path.

Our method analyzes the data stream of each execution path of the program synthetically. Based on an output variable with basic expectation, dependencies analysis will be made between reverse data variables respectively. Then confirm data dependencies of statements and irrelevant statement set for each path. Irrelevant statement set represents all statements in this collection are not related to other statements for this execution path, and they are irrelevant and can be ignored. We can delete the irrelevant statement set of each path, incorporate new execution paths without irrelevant sets into a complete data flow diagram of execution paths, and get a new program slicing of small code amount. And then analyze the data flow diagram to full data dependencies, according to coverage based fault localization. Count failure and successful execution frequency of each node, and calculate the defect tendency values. Sort the values, and ultimately locate the defect in a statement exactly in the program.

In this way, the granularity of location is the basic code statement. Compared to only control flow based fault localization, positioning granularity is reduced to a code statement instead of the basic code block. Under the premise of no loss the degree of automation of location, it improves the accuracy of location.

Combining data flow analysis with coverage based fault localization, it will improve the accuracy of fault localization and reduce the granularity of fault localization on data variables level. And it’s more convenient to debug for programmers.

The basic idea of the study is shown in Figure 2 as follows:

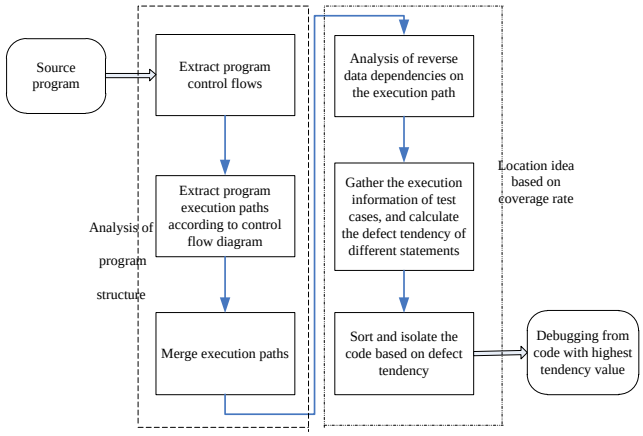


Figure 2. Block diagram of CPD research idea

Extract program execution path: Control flow analysis mainly extract various execution paths of the program according to the execution of test cases, combining relevant gcc commands.

Merge execution paths: Integrate the streamlined execution paths into a complete dependence graph of the program, to facilitate the follow-up statistical analysis of coverage.

Analysis of reverse data dependencies on the execution path: This is the main work of reverse data dependencies model. Extract the data dependencies on the execution path by analyzing the text of program.

Calculate the defect tendency: Quantify calculation model by combining the statistical error correction, and realize the calculation of different code statements defect tendency by programming.

Sort and isolate the code based on defect tendency: Sort the values of defect tendency from high to low, and then make the isolation.

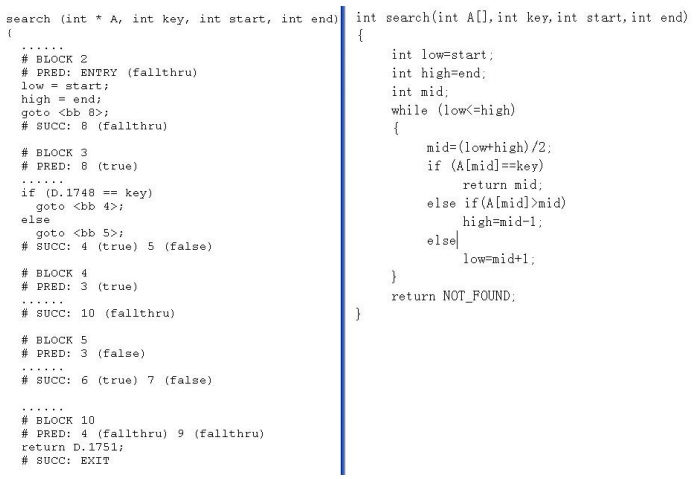
4. Extraction of Execution Paths

The basis of reverse data dependencies is extracting different execution paths of the program. But the basis of extracting execution paths is extracting control flow. So we will introduce the content about extracting control flow in the following subsection.

(1) Extraction of control flow

For the follow-up study convenience, in the extraction of the program control flow, we need information as follows: starting line number, previous node and successor node of each basic code block.

To obtain program control flow graph, there are few specialized tools to extract the program control flow graph in existing tools. We found that an option -fdump-tree-cfg-blocks in the gcc compiler commands can be used to generate the control flow of the program. Comparison for the program after the command processing and the original program code is shown in Figure 4.



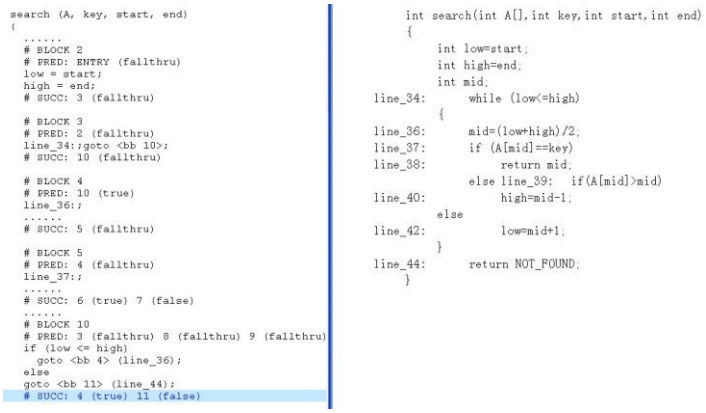
```
search (int * A, int key, int start, int end)
{
  ......
  # BLOCK 2
  # PRED: ENTRY (fallthru)
  low = start;
  high = end;
  goto <bb 8>;
  # SUCC: 8 (fallthru)

  # BLOCK 3
  # PRED: 8 (true)
  ......
  if (D.1748 == key)
    goto <bb 4>;
  else
    goto <bb 5>;
  # SUCC: 4 (true) 5 (false)

  # BLOCK 4
  # PRED: 3 (true)
  ......
  # SUCC: 10 (fallthru)

  # BLOCK 5
  # PRED: 3 (false)
  ......
  # SUCC: 6 (true) 7 (false)

  ......
  # BLOCK 10
  # PRED: 4 (fallthru) 9 (fallthru)
  return D.1751;
  # SUCC: EXIT
```

```
int search(int A[], int key, int start, int end)
{
    int low=start;
    int high=end;
    int mid;
    while (low<=high)
    {
        mid=(low+high)/2;
        if (A[mid]==key)
            return mid;
        else if(A[mid]>mid)
            high=mid-1;
        else
            low=mid+1;
    }
    return NOT_FOUND;
}
```

Figure 3. Comparison of the compiled and the original program code

The figure shows, each function defined in the program is divided into different basic code blocks by the compiler option. And specific markers present the control relationships between basic code blocks of each function. It is relatively easy to get the program control flow graph by text analysis of the compiled file. But there isn't starting line information of basic code blocks in the control flow graph gained above. If we don't know the code line number, we will not learn the coverage of each basic code block, from the cov file containing the implementation of the program. So we must preprocess the source code first, inserting a line number tag at the beginning of each code block (called marking), then use gcc-fdump -tree-cfg-blocks to obtain the control flow.

What program marking do is inserting a line number into the starting line of each basic code block. We can use labels allowed in the program to make it. Such as add "Line_num:" to the beginning of this line, "num" in which is the line number. It should be paid attention that some lines can't be marked. For example, insert a label into a line defining a variable, what will makes gcc compile unsuccessfully, then we won't get the control flow expected. In this case, we need to skip the line defining variables and mark labels to the line operating variables actually. The comparison between marked source code and code compiled by command gcc-fdump -tree-cfg-blocks is shown in Figure 5.



```
search (A, key, start, end)
{
  ......
  # BLOCK 2
  # PRED: ENTRY (fallthru)
  low = start;
  high = end;
  # SUCC: 3 (fallthru)

  # BLOCK 3
  # PRED: 2 (fallthru)
  line_34:;goto <bb 10>;
  # SUCC: 10 (fallthru)

  # BLOCK 4
  # PRED: 10 (true)
  line_36:;
  ......
  # SUCC: 5 (fallthru)

  # BLOCK 5
  # PRED: 4 (fallthru)
  line_37:;
  ......
  # SUCC: 6 (true) 7 (false)
  ......
  # BLOCK 10
  # PRED: 3 (fallthru) 8 (fallthru) 9 (fallthru)
  if (low <= high)
    goto <bb 4> (line_36);
  else
  goto <bb 11> (line_44);
  # SUCC: 4 (true) 11 (false)
```

```
int search(int A[], int key, int start, int end)
{
    int low=start;
    int high=end;
    int mid;
line_34:    while (low<=high)
    {
line_36:    mid=(low+high)/2;
line_37:    if (A[mid]==key)
line_38:        return mid;
            else line_39:  if(A[mid]>mid)
line_40:        high=mid-1;
    else
line_42:        low=mid+1;
    }
line_44:    return NOT_FOUND;
}
```

Figure 4. Comparison of the marked source code and compiled code

(2) Extraction of execution paths

Compare the program control flow graph above with the execution results of the related test cases; we can know that the execution of each test case corresponds to an execution path, failure test cases correspond to failure execution paths. In fact, extractions of execution paths by programming do the text analysis for .cfg file. The result is, line numbers of codes run in this execution are extracted by execution order in turn, and then these numbers are stored in a structure linked list.

What the contents of this section above describe is the algorithm extracting an execution path. Extract for each failure execution path based on this algorithm, and then collect line numbers of codes extracted and compute, and a failure execution flow diagram will be got. Next, extract the data dependencies of the failure execution flow diagram based on the reverse data-dependent model proposed in 3.2.

(3) Data preprocessing

Sort basic code blocks in the failure execution flow diagram by execution order. To every error, use the corresponding gcov file (coverage rate) with executing the test case successfully and unsuccessfully, to get the information of the test case covering basic code blocks. Then make a sheet to record, in the form shown in Table 2. Each row in the table represents the situation of a test case covering basic code blocks in the program, and 0 indicates that the basic code block is not covered by the test case, 1 is. We will make tables respectively for all the correct execution test cases and failure execution test cases of each error. For instance, program flex we used to experiment has 19 errors, so there is a total of 19 groups of tables. Each group contains a table of basic code blocks coverage, which is corresponded by executing test cases successfully and unsuccessfully, as Table 1 shows.

Table 1. Failure execution test cases covering basic code blocks

No.of basic code blocks	1	2	3	4	5	6	……	M
Testcase1	1	1	0	1	1	0	……	1
Testcase2	1	0	1	1	1	0	……	1
:	1	0	1	1	1	0	……	1
:	1	1	0	1	0	1	……	1
Testcase(N-1)	1	0	1	1	1	0	……	1
TestcaseN	1	1	0	1	0	1	……	1

By analyzing, this paper argues that it's more significant that N failure executions distribute in the N different control flows than in a control flow path. In other words, for the same control flow statistics of different executions is just double counting. Only for different control flow paths, it will have a positive effect on solving program failure paths.

Therefore, considering a plurality of execution paths will be obtained in the actual testing process, after data preprocessing this paper will merge all execution paths on the basis of corresponding control flows.

5. Experimental Results and Analysis

In order to verify the effect of the proposed method, this section will take advantage of method CPD proposed in this paper to locate the error in project flex v1. The result will be compared with the results obtained by using SBI, Tarantula, and CT.

There are totally 19 faults in project flex v1 to be experimented. Among them, the position of 8, 12, 13, 16, 18 faults, can't be located regardless of which method to be used. The main

reason is programs with 12,13,16,18 fault crash when running failure execution test cases. And the fault position of 8th error is in the macro definition. So it's unable to obtain the code coverage information of the fault position in both cases, and cause coverage based fault localization method invalid. Table 2 shows the remaining 14 faults, including the fault number, fault ID, and fault type. Fault types are divided into three: condition faults of if conditional statement and while loop statement, assignment faults in the assignment and calculation, function fault in the function call process.

Table 2. Faults classifications of project flex v1

Fault number	ID	Fault type
1	FAULT_F_TW	Conditional statement fault
2	FAULT_F_FW	Function fault
3	FAULT_F_KL	Assignment fault
4	FAULT_F_KL	Assignment fault
5	FAULT_F_FW	Function fault
6	FAULT_F_KL	Assignment fault
7	FAULT_F_KL	Assignment fault
9	FAULT_F_TW	Conditional statement fault
10	FAULT_F_KL	Assignment fault
11	FAULT_F_TW	Conditional statement fault
14	FAULT_F_TW	Conditional statement fault
15	FAULT_F_FW	Function fault
17	FAULT_F_FW	Function fault
19	FAULT_F_FW	Function fault

The number of conditional judgment fault is 4: 1, 9, 11, 14. The number of assignment fault is 5: 3, 4, 6, 7, and 10. The number of function call fault is 5: 2, 5, 15, 17, and 19.

To judge the CPD's location effect, we need to contrast the result with location methods existing. The code to be detected is less and the number of fault to be located is larger, then the location method is better. We chose three location methods, Tarantula, CT, and SBI.

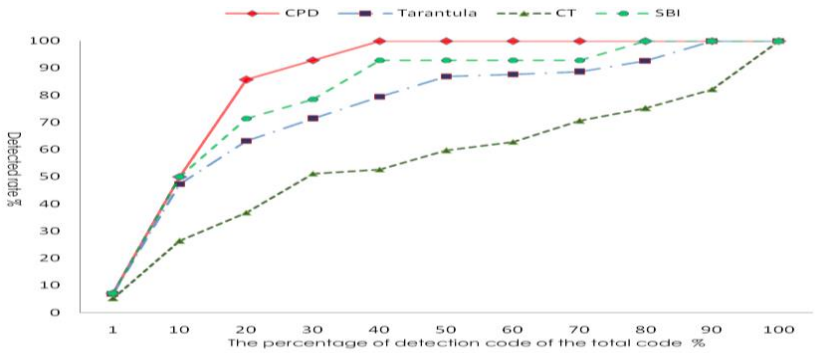


Figure 5. The location effect for various location methods

From the Figure 5, we can see, in four location methods, the corresponding curve of the CPD method is always higher than the other curves, that is the location effect of CPD method is best. If the amount of code to be detected is 1% of the total, CPD and SBI will capture 7.14% of the faults, other two less than 7.14%.

6. Conclusion

In this paper, by analyzing the advantages and disadvantages of the more generally applicable CBFL and program slicing method, for shortcomings, we proposed a new fault localization method CPD while retaining the advantages. This article needs to do further research in the following areas: how to reduce the cost of data dependencies extracted and how to further improve the accuracy of fault localization without loss of the automated process.

Acknowledgements

This research was partially supported by the National Basic Research Program of China (973 Program) under grant No. 2011CB302605 and the National Science Foundation of China (NSF) under grants No. 61173145 and No. 61202457.

References

[1] C. Shen, H. Zhang, D. Feng, Z. Cao and J. Huang, "Summary of Information Security", Science in China (Chinese version), vol. 37, no. 2, **(2007)**, pp. 129-150.
[2] B. Fang, T. Lu and C. Li, "Survey of software assurance", Journal on Communications (China version), vol. 30, no. 4, **(2009)**, pp. 106-117.
[3] M. Weiser, "Program Slicing", In IEEE Transactions on Software Engineering, vol. SE-10, no. 4, **(1982)**, pp. 352-357.
[4] T. Chilimbi, B. Liblit, K. Mehra, A. Nori and K. Vaswani, "Holmes: effective Statistical Debugging via Efficient Path Profiling", In Proceedings of ICSE 2009. IEEE Computer Society Press, Los Alamitos, CA, **(2009)**.
[5] M. B. Swarup and P. S. Ramaiah, "A Software Safety Model for Safety Critical Applications", IJSEIA, vol. 3, no. 4, **(2009)** October, pp. 21-32.
[6] P. R. Srivastava and T. -h. Kim, "Application of Genetic Algorithm in Software Testing", IJSEIA vol. 3, no. 4, **(2009)** October, pp. 87-96.
[7] M. Beldjehem, "A Unified Granular Fuzzy-Neuro Framework for Predicting and Understanding Software Quality", IJSEIA, vol. 4, no. 4, **(2010)** October, pp. 17-36.

Authors

Hui He

Hui He received the B.S., M.S. and Ph.D. degree in computer science from Harbin Institute of Technology, Harbin, China. Since September 1999, she has been with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China, where she became an Associate Professor in October 2007. Her research interests include network computing, network security.

Lei Zhao

Lei Zhao, Master, in computer science from Harbin Institute of Technology, Harbin, China. Her research interests include network security, software engineering.

Qiao Li

Qiao Li, Ph.D. Student, in computer science from Harbin Institute of Technology, Harbin, China. His research interests include network security, software engineering.

Weizhe Zhang

Weizhe Zhang, professor and Ph.D., supervisor in the school of Computer Science and Technology at Harbin Institute of Technology. He obtained the PhD degree in Computing Science from the Harbin Institute of Technology in China. He conduct research in parallel and distributed system, cloud computing, real-time computing and computer network.

Dongmin Gao

Dongmin Gao, Master, in computer science from Harbin Institute of Technology, Harbin, China. Her research interests include network security, software engineering.