

## Software fault debugging based on data flow analysis

Xi Guo<sup>1,a</sup>, Pan Wang<sup>2,b</sup> and Peng-Fei Wu<sup>3,c,†</sup>

<sup>1</sup>Department of Computer Science, College of Informatics,  
Huazhong Agricultural University, Wuhan 430070, China

<sup>2</sup>Wuhan Electric Power Technical College, Wuhan 430079, Hubei, China

<sup>3</sup>Department of Computer Application, College of Informatics,  
Huazhong Agricultural University, Wuhan 430070, China

<sup>a</sup>xguo@mail.hzau.edu.cn, <sup>b</sup>wangpan6712063@163.com, <sup>c</sup>chriswpf@mail.hzau.edu.cn

\*Corresponding author: Peng-Fei Wu

Data Flow Analysis is a difficult issue in the domain of fault localization, and many software faults are related to the information of data flow. The dependency between the variants and the define-use chain are discussed in this paper, and trace the impact to the variants in the process of operation. In this paper, a data flow model is proposed which can demonstrate the change of the variant value and the dependency between the variants and it can be used to debug the faults in the program. Experimental results demonstrate that this method has better results than the traditional methods.

*Keywords:* Software debug; Data flow analysis; Software testing.

### 1. Introduction

There are many kinds of software faults in the domain of software analysis, and the fault relate to data flow is very important. Researchers have been working on this theme for a long time, but current researches rarely focus on the aspects of variant values and context information and so on, what is more, this information is of vital importance to the analysis of fault localization.

In this paper, a data flow model is proposed which can monitor the change of the variants and their dependency, and this can be used to debug the faults. The degree of the relationship between the variants and the transition from one variant to another are calculated both in success and failure testing.

At first, the fault localization via variant information is shown, and then definition of data flow analysis is shown, and next, the method of fault localization is discussed, the experiment results and future works are demonstrated at last.

### 2. Data Flow Analysis and Fault Localization

The value of the variants can be changed from the operation of the users, and the data flow of the variants consists of the changes to the variants. The operation to

the variants includes definition and use. The data flow analysis is the method which can discover the incorrect definition and use of the variants, and record the real time value to check the program states.

The coverage of use-define chain is used to debug the faults [1], which calculates the degree of faults between each use-define chain. Delta debugging forms the state chart by obtaining all the variants and their values[2], and then find the variants that lead to the failure by observe the difference between the correct and fault executions according to the sub chart. In order to improve the efficiency of fault analysis and localization, some research focus on the program spectrum that can be utilized to fault localization. The types of fault are studies, and the process of Markov is used to predict the failure, and the select the proper method to tackle the problem [3].

### 3. Data Flow Model

For a program  $S$ , its variant set  $V$  can be shown as  $V = \{v_1, v_2, \dots, v_n\}$ .  $V$  can be shown as  $V = V_c \cup V_r$ , where  $V_c$  is the variants that created in the execution of the program, and  $V_r$  is the variants that created in return process after function calls. The “use” of variant refers to the reference of the variant, and its value is accessed, but cannot be changed. The “define” of variant means the value is modified. A variant would be defined several times during the execution of a program, but its value may not be changed. Thus, it would be difficult to distinguish the impact from different locations. A variant can be modified by several code sections, so it needs to find the states of the variant during execution.

The status of a variant can be defined as  $P \times V \times N$ , where  $P$  is the statements of the target program, and  $V$  is the variants during the execution of the program,  $N$  records the times of the definition operation. Thus  $(l, v, n)$  is a tuple of  $P \times V \times N$ , which means the program modifies the definition of variant  $v$  in location  $l$  as the  $n^{\text{th}}$  since the execution of the program. For any two states  $(l_1, a, n_1)$  and  $(l_2, b, n_2)$ , if they have the same value during the execution, that means  $a$  and  $b$  are the same variants, and there is no definition between the two statements. If  $n_1$  is less than  $n_2$ , the source of the transition is  $(l_1, a, n_1)$  and the target is  $(l_2, b, n_2)$ , and vice versa.

The dependency between variants is defined as follows: if  $y$  is accessed in define operation of  $x$ ,  $y$  is depended on  $x$ . The relation between variants reveals the impact from other variants. Let  $d$  be the dependent function[4]:  $P \times V \times N \rightarrow H(P \times V \times N)$ . For two define operations  $(l_1, a, n_1)$  and  $(l_2, b, n_2)$ , if  $(l_1, a, n_1) \in d(l_2, b, n_2)$ , which means  $(l_2, b, n_2)$  is dependent on  $(l_1, a, n_1)$ . The dependency between  $a$  and  $b$  can be defined as follows:

$$\text{Dependency}(a, b) = \{((l_1, a, n_1), (l_2, b, n_2)) | ((l_1, a, n_1) \in d(l_2, b, n_2))\}$$

Where  $(l_1, a, n_1)$  is the precedent of the dependency, and  $(l_2, b, n_2)$  is the descendant of the dependency. Variant can also be self-dependent, and there are two kinds of dependent relations: the first kind is for different variants, such as  $z = x + y$ , where  $z$  is dependent on  $x$  and  $y$ . The second kind is self-dependent, such as  $x = x + 1$ , where the value of  $x$  generate dependent relation, that is the current state is dependent on the previous one. Thus, this kind of self-dependent can be defined as follows:

$$Dependency(a, a) = \{((l_1, a, n_1), (l_2, a, n_2)) | ((l_1, a, n_1) \in d(l_2, a, n_2))\}$$

This is a special program dependent relation, and usually  $Dependency(a)$  is short for  $Dependency(a, a)$ .

The define operations in each variants and their dependent relations among them forms a sequence, which is used to define the transition-dependency relation. For a certain execution, the referred variants set is  $V = \{v_1, v_2, \dots, v_n\}$ , the route of the variant state transition can be defined as  $route(V)$ , which is formally shown as:

$$route(V) = route(v_1) \cup \dots \cup route(v_n)$$

The dependent relation in the states of variant definition can be defined as  $Dependency(V_1, V_2)$ , and  $Dependency(V_1, V_2)$  can be obtained as follows:

$$Dependency(V_1, V_2) = \bigcup_{v_1 \in V_1, v_2 \in V_2} Dependency(v_1 v_2), \text{ where } v_1 \in V_1, v_2 \in V_2$$

The data flow route  $DFR(V)$  can be expressed as follows:

$$DFR(V) = (Node, Edge)$$

Where  $Node$  is the set of vertex:  $Node = \{(l_1, x_1, n_1), \dots, (l_n, x_n, n_n)\}$ , and  $Edge$  is the set of edges:  $Edge = \{(a, b) | (a, b) \in route(V_1) \parallel (a, b) \in route(V_2)\}$ .

During the process of data analysis, static and dynamic analysis methods are combined to ensure the accuracy. The static analysis method is used to collect the variants and statements, and the use and define information over these variants, and the control flow graph(CFG) can also be generated. The dynamic method can monitor the trace of the execution of the target variants.

#### 4. Fault Debugging Method Based on Data Flow Information

After the trace collection of the program execution, the next step is to find out the exact functions that may lead to failure. These target functions may exist both in one class and in many classes. Usually,

the route of success execution and that of the failure execution are not the same.

According to check whether there are direct precedents nodes both in the route of the success and failure executions, the nodes in the data flow model can be divided into 3 parts:

(1) The current node has the dependent relation to the precedent node. The node  $V$  has a direct precedent node, and their relation is dependency. The conditional probability under  $V = V^n$  is:

$$\frac{p(N(V = V^n | Precedent(V) = V_i))}{N(Precedent(V) = V_i)}$$

Where  $Precedent(V)$  is the set of precedent nodes.  $(N(V = V^n | Precedent(V) = V_i))$  is the existing times under condition of  $V = V^n | Precedent(V) = V_i$  and  $N(Precedent(V) = V_i)$  is the existing time under condition of  $Precedent(V) = V_i$ .

(2) The current node has direct precedent node, and the edge between them is the type of transition. W The conditional probability under  $V = V^n$  is:

$$p(V = V^n) = \frac{N(V = V^n)}{N(V)}$$

Where  $N(V = V^n)$  is the existing number of times under  $V = V^n$ , and  $N(V)$  is the existing number of times under  $V$ .

(3) The current node has no direct precedent node, and the calculation method is the same as the former one.

After collecting the operation set of suspect variants, it would be order the state of the variants by the degree of the suspicion, and it is possible to locate the failure statements since there are line numbers in the operation of program static analysis.

## 5. Experiment and Analysis

A demo is developed to verify the proposed method, the static analysis method analyze the control flow, and generate the control flow graph, and find all the possible defined variants and used variants, and then find all the use-define chain. Meanwhile, the execution details, including line number and so on are recorded, which can help to the dynamic analysis.

A group of benchmarks are selected to compare the efficiency of the proposed method. Some program are implanted several failures, which are related to the data flow information, the details of these benchmarks are listed Table 1.

We compare our method to the traditional use-define chain method. Since NalXml(V1) has the greatest number of failures, our experiment select is as the benchmark. The experiment results are show in Table 2.

Table 1 Testing programs

Program	Functions	Line number	Test cases	Number of failures
SortUtil	sorting	513	9	2
NalXml(V1)	Context analyzer	3391	206	3
NalXml(V2)	Context analyzer	3507	267	5
Xml_security(V1)	Context security protocol	20921	108	5
Xml_security(V2)	Context security protocol	21973	117	3
Diff	Context compare	729	62	2

Table 2 Comparison of the methods

Fault label	Use-define Chain		Our method	
	Time consumption(s)	Degree of suspicious	Time consumption(s)	Degree of suspicious
Fault 1	3.5	0.31	4.9	0.52
Fault 2	4.6	0.57	9.3	0.66
Fault 3	19.5	0.63	25.5	0.71
Fault 4	15.1	0.47	21.9	0.57
Fault 5	23.5	0.29	35.7	0.43

The table above illustrates that our method consume more time than use-define chain method, but our method can get a higher degree of suspicious, which can help the researchers to debug the program more precisely and efficiently.

## 6. Conclusion

The faults about program control logic are related to the values of the variants, and the changes to the variants also affect their dependency. Thus, the program variants include uncertain features. In the future, we will continue to explore our method, and try to improve the efficiency and time consumption and so on.

## Acknowledgement

This project is supported by the Fundamental Research Funds for the Central Universities under grant No. 2662015QC009 and the National Science Foundation of China (NSFC) under grant No. 61502194.

**References**

- [1]. Santelices R, Jones JA, Yu YB, Harrold MJ. Lightweight fault-localization using multiple coverage types. In: Proc. of the IEEE 31st Int'l Conf. Washington: IEEE Computer Society, 2009. 56–66.
- [2]. Zeller A. Isolating cause-effect chains from computer programs. In: Proc. of the 10th ACM SIGSOFT Symp. on Foundations of Software Engineering. New York: ACM Press, 2002. 1–10.
- [3]. Zhang YQ, Zheng Z, Ji XH, Zhang WB, Zhang ZY. Markov model-based effectiveness predicting for software fault localization. *Jisuanji Xuebao*, 2013,36(2):445–456(in Chinese).
- [4]. Yang B, Wu J, Liu C. Software fault localization based on data chain. *Ruan Jian Xue Bao/Journal of Software*, 2015, 26(2):254–268(in Chinese).