Improving Fault Localization through Fault Propagation Analysis

Zhimin Fang[1], Danni Li[1], Shifei Chen[2] and Rong Chen[1]
[1] Dalian Maritime University, Dalian 116026, China
[2] Sichuan University - Pittsburgh Institute, Chengdu 610207, China

Abstract—**Lots of studies have been done to pinpoint program faults almost to the exact line where different types of abnormal behaviors occur. Among them there is the promising testing-based fault localization (TBFL) technique, the aim of which is to locate the faults in the program automatically when the program is executed with test suite. This paper proposes an improved TBFL method augmented with fault propagation. It localizes faults in three major steps: generate the execution path and dependency information of program, modify the existing fault state of program, and ultimately infer statement suspicious score. Our empirical results show that, for the subject we studied, our technique can get better effectiveness than others. Empirical results show that our method can help to debug efficiently and get better effectiveness than other competitors.**

Keywords-software debugging; fault localization; fault propagation; program dependency.

I. Introduction

Debugging software error is always a time-consuming and costly problem for software developers to solve [1]. Locating faults in programs usually cost most of time in the debugging process. Clearly, techniques that can reduce the time required to locate faults can have a significant impact on the cost and quality of software development and maintenance [3].

The theory of automated fault localization has been presented and lots of researchers devote their energies to this field. TBFL is one of the most promising fault localization techniques, which ranks suspiciousness of statements by counting their occurrences in failing and successful runs of a buggy program. Kai Yu et al proposed a spectrum-based fault localization (SFL) technique named LOUPE [5] which uses multiple spectra-specific models to locate faults with the dynamic dependence information. Sun, Xiaobing, et al. propose an iterative fault localization process of selecting test cases for effective fault localization, which identifies as many faults as possible in the program until the stopping criterion is satisfied [15]. Shu T, Ye T, et al. propose a fault localization method based on statement frequency [2].

The dynamic dependence information, which contains data and control dependence, describes the relationship between the variables by data dependence and the statements by control dependence. We observe that the programmers usually figure out the faulty statements by looking into how the infection is propagated along the executed path and the dynamic dependence information. So it is a good idea to introduce fault propagation information to improve fault localization, which could improve the precision of the fault localizer and save time.

In this paper, we present the Fault Propagation based Fault Localizer (fpFL) which establishes fault propagation model to pinpoint the fault by combining the suspicious score of each node and the factor of fault propagation. We calculate the suspiciousness of a statement by bigger one of the two scores with control and data dependence models. Finally we can get the rank list of all the suspicious statements and the faults are put in the obvious place to assist fault localization.

Our empirical experiment indicates that our technique is useful and it can improve the locating effect. The main contributions of the paper are twofold: (1) A presentation of an approach that build a fault propagation model with control to evaluate each statement and produces the effective rank list of suspicious statements with our algorithm. (2) The results of a set of our empirical studies, with our technique and some other techniques. The studies evaluate the effectiveness of our technique with the Siemens Test suite.

In the next section, we present our technique and illustrate it with a motivating example.

II. A Motivating Example

In this section, we briefly review the control and data dependences. Then, we will use a common example to propose the problem and draw out our technique. Program dependence graph is useful in software engineering applications, such as testing [5], fault localization [6], [7] and model potential semantic dependences [8]. A program dependence graph represents control dependence in a program. Control dependences are typically defined in terms of control flow graphs. Data dependences are typically defined in terms of data flow graphs. They can capture the effects of predicate statements and data interactions on program behavior. They can capture the abnormal behavior through control and data flow if the program has bugs. If a bug is propagated by control flow, it is easily involved the abnormal behavior though control dependences. Similarly a bug propagated by data flow is easily involved the abnormal behavior though data dependences. All in all, different fault types are influenced by different dependences.

First, we use one example of program is_num_constant() to illustrate our motive of fault localization. The code checks the input string whether it is a constant number, which shows a function extracted from the faulty version v6 of program print_tokens2 and a test suite in the Siemens suite [10]. It is shown in Figure 1 and Figure 2.

Fault: s4 should be **ch=*(str+i);**			Test cases(6)					
			1	2	3	4	5	6
Int is_num_constant(token str){								
char ch;								
int i=1;	S1	b1	•	•	•	•	•	•
if (isdigit(*str)){//if	S2		•	•	•	•	•	•
while(*(str+i)!='\0'){	S3	b2		•	•	•	•	•
ch=*(str+i+1);	S4	b3		•	•	•	•	•
if(isdigit(ch))	S5			•	•	•	•	•
i++;	S6	b4			•		•	
else		b5						
return(FALSE);	S7			•	•	•	•	•
} /*end WHILE */								
return(TRUE);	S8	b6						
}//end if								
else		b7						
return(FALSE);	S9		•					
}			P	F	P	P	F	P

FIGURE I. EXAMPLE FAULTY PROGRAM (LEFT); TEST SUITE, AND TEST RESULTS (RIGHT)

In Figure 1, the left part is the program is_num_constant(), which has a fault in line 7, the right part is the test suite with 4 passed test cases and 2 failed test case, each test case occupies a column.

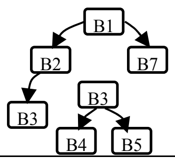
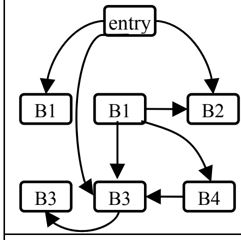
int is_num_constant (token str) Control and Data dependences		Test cases(6)					
		1	2	3	4	5	6
Control Dependences 	(b1,b2)		•	•	•	•	•
	(b2,b3)		•	•	•	•	•
	(b3,b4)			•		•	
	(b3,b5)		•	•	•	•	•
	(b1,b7)	•					
Data Dependences 	(entry,b1)	•	•	•	•	•	•
	(entry,b2)		•	•	•	•	•
	(entry,b3)		•	•	•	•	•
	(b1,b2)		•	•	•	•	•
	(b1,b3)		•	•	•	•	•
	(b4,b3)			•		•	
	(b3,b3)		•	•	•	•	•
	(b1,b4)			•		•	
		P	F	P	P	F	P

Figure II. THE CONTROL DEPENDENCE GRAPH AND SPECTRA OF IS_NUM_CONSTANT()

We record a small back dot at the intersection of the statement row and the test case column when the statement is executed, the results are given at the bottom of the test cases column, "P" represents the result is passed and "F" represents failed, such as the test case t5, the program executed the statement s1, s2, s3, s4, s5, s6, and s7, and the result is failed. In the Figure 2, the control dependence of the program is_num_constant() is shown in Figure 2.

In Figure 2, the nodes represent the relatively independent block, which is inseparable and compose of continuous sentences. The edges represent the dependency relationship between two blocks. We record a small back dot at the intersection of the dependency pair row and the test case column when the edge is executed, the results are also given at the bottom of the test cases column. For example, the back dots in first row represent the dependency pair (b1,b2) is executed in the test {2,3,4,5,6}. According to Figure 2.

III. METHODS

A. *Fault Localization Model*

To illustrate our method, given a passed test suite T={t1,t2,t3, ⋯ tm}, and a failed test suite T'={t1',t2',t3', ⋯ tm'} for a program P. In the program is_num_constant(), passed test suite T={t1 ,t3 ,t4, t6}, T'={t2 ,t5} is the failed suite. Give the B={b1,b2,b3,⋯,bn} as the set of basic block, a statement set S={s1 ,s2 ,s3, ⋯,sn}, sk(1≤k≤n) is a statement in P. Give a dependence pair ε(bi,bj), which means bi has a control dependence on the bj which is the successor node of bi. We define ε(*,bi) as all dependence pairs that dependent on the bi as the successor node. In the program execution process, it will be executed to a number of dependency pairs, so we define θ(ε,tk) as whether the program cover the dependence pair ε in execution of test cases tk, if true the value is 1, else the value is 0. In the program is_num_constant(), ε(s2,s3) is cover in the test 2, so θ((s2,s3),t1)=1.

With the spectra of dynamic control dependence, we can measure initial suspicious score of each node. There are several similarity formulas to evaluate the suspicious of the node. we choose Ochiai [9] formula as our metrics. The initial suspicious score of a node measures as follow:

$$sus(b) = \frac{b_{ef}}{\sqrt{(b_{ef}+b_{nf})(b_{ef}+b_{ep})}} \quad (1)$$

where b belongs to the set of basic block. In (1), bnf refers to the number of test cases that b have not been executed in the failed test case. bef and bep refer to the number of test cases that block b have been executed in the failed test case and the passed test case, so bef+bep represents the number of test cases that the basic block B appears in its execution path. This value is used as the initial probability of the algorithm. Then we need to consider the impact of fault propagation on the node. Define θ√(ε) and θ×(ε) as the coverage proportion of the dependence pair ε in T and T'. The corresponding formula are given in (2):

$$\theta^{\surd}(\varepsilon) = \frac{1}{u}\sum_{t_k \in T}[\theta(\varepsilon,t_k)] \quad \theta^{\times}(\varepsilon) = \frac{1}{v}\sum_{t'_k \in T'}[\theta(\varepsilon,t'_k)] \quad (2)$$

where the u and v represent the number of successful test cases and failed test cases respectively, tk and t'k belongs to T and T' respectively. As mentioned above if the dependence pair ε is executed in a test case tk , θ(ε,tk) equal 1, or 0. In doing so, we can calculate the successful and failure coverage proportion of each dependence pair ε. If a dependence pair is higher on the frequency of the test cases that appear in the error, the more likely it is to go wrong or the more likely it is to appear on the propagation path. Then we need to quantify this possibility to measure the suspicious of the dependence pair. The metrics is shown as following:

$$\theta^{\triangle}(\varepsilon)=\frac{\theta^{\times}(\varepsilon)-\theta^{\surd}(\varepsilon)}{\theta^{\times}(\varepsilon)+\theta^{\surd}(\varepsilon)} \tag{3}$$

The formula (3) is compared in dependence on the success and failure in the test case coverage rate to calculate the suspicious of the dependence pair ε. When the value of $\theta^{\triangle}(\varepsilon)$ is positive, that the coverage proportion of ε in failure test case is greater than it in the successful test case, so the dependence pair ε in the failed execution has a higher frequency, which indicates that the more closely it related to program error. When value equals 0, which the correlation degree between ε and error is lower than the one which its value is positive. Similarly, when the value is negative, the correlation degree between ε and the error is the lowest. Then we need to calculate the propagation rate of each dependence pair.First we introduce a concept w(ε) as the propagation rate proportion size of the ε. The propagation proportion calculation method, as shown in the formula (4).

$$W\left(b_j,b_k\right)=\frac{\theta^{\triangle}(b_j,b_k)}{\sum_{\forall(*,b_k)}[\theta^{\triangle}(*,b_k)]} \tag{4}$$

where $\theta^{\triangle}$(bj,bk) is the suspicious of the dependence pair ε(bi,bj), and $\theta^{\triangle}$(*,bk) represent that the sum of the dependence pairs' suspicious score whose successor node is bk as mentioned above.

```
Algorithm: FPFL
Input: dependencies E, statement S, block B,
       pass or fail result of $t_i \in T \cup T'$
Output: the suspicious score rank list of statements
foreach: $b \in B$
    $b_{ef}$ ← number of b appear in the fail test cases
    $b_{nf}$ ← number of b not appear in the fail test cases
    $b_{ep}$ ← number of b appear in the pass test cases
    $sus(b) \leftarrow b_{ef} / \sqrt{(b_{ef}+b_{nf})(b_{ef}+b_{ep})}$
foreach: $\varepsilon \in E$
    $\theta^{\surd}(\varepsilon)$ ← coverage rate of ε in pass test cases
    $\theta^{\times}(\varepsilon)$ ← coverage rate of εappear in fail test cases
    $\theta^{\triangle}(\varepsilon) \leftarrow (\theta^{\times}(\varepsilon)-\theta^{\surd}(\varepsilon))/(\theta^{\times}(\varepsilon)+\theta^{\surd}(\varepsilon))$
foreach: $(b_j,b_k) \in \varepsilon$
    $W(b_j,b_k) \leftarrow \theta^{\triangle}(b_j,b_k)/\sum_{\forall(*,b_k)}[\theta^{\triangle}(*,b_k)]$
foreach: $b_j$ exists in ε
    if $b_j$ is not a exit block then
        $suspicious(b_j) \leftarrow sus(b_j)+\sum_{\forall(b_j,b_k)}[suspicious(b_k)\cdot W(b_j,b_k)]$
    else
        $suspicious(b_j) \leftarrow sus(b_j)$
foreach: $s_i \in S$
    if $s_i \in b_j \wedge b_j$ exists in $\varepsilon_t$
        $suspicious(s_i) \leftarrow suspicious(b_j)$
    else
        $sus_t(s_i) \leftarrow 0.00$
statement rank list L ← sort statements by $suspicious(s_i)$
return L
```

FIGURE III. FpFL ALGORITHM

In formula (4), the numerator represent the suspicious score of ε(bi,bj) and the denominator represent the sum of the dependence pairs' suspicious score whose successor node is bk. The ratio of the formula (4) represent the impact of bj on bk. In particular, when the denominator is 0, we set it to a minimum of 10^{-10} in order to ensure the correctness of the calculation. After obtaining the program's propagation, we can calculate the suspicious degree of each node according to the relationship between the propagation rate and the node. Then we combine the impact of other nodes calculated by (4) and the suspicious score of the node itself calculated by (1) to obtain the suspicious score of each block, the formula is shown in (5).

$$suspicious\left(b_j\right)=sus(b_j)+\sum_{\forall(b_j,b_k)}[suspicious(b_k)\cdot W(b_j,b_k)] \tag{5}$$

where sus(bj) has been calculated as the suspicious score of node bj itself in (1), and the latter part is the product of the ε(bi,bj)'s propagation rate and the suspicious score of node bk. In doing this, we can obtain a set of equations about the node suspicious degree, and the suspicious degree of the leaf node is the suspicious score of the node itself. So the n element equation group can be solved by the Gauss's elimination method. For each statement in the block is the same, we can get a suspicious degree ranking table of each statement. According to the table we can finally pinpoint program fault. Our method algorithm for fault localization has been given in Fig. 3.

IV. EXPERIMENTS

A. *Experiment Setup*

In our studies, we use the Siemens suite to analysis our techniques. It consists of 7 programs which consist of correct code, several versions of faulty code, thousands of test cases and others. In our experiment, we use 122 versions for our studies shown in Table 1, because some versions could not be instrumented with our tool, some versions' fault lines exist in the head files, we remove those versions in our experiment. We run all the given test cases listed in Table 1, but some of them can not get the output with specific version so we delete them in our studies.

After estimating the suspicious score of each executed statement in the program, we sort the scores in decreasing order to help the programmers to find bug. Some statements may have the same scores in specified precision. We set their rank values with the last one in the group statements. Take s6 and s7 for example in Fig. 1, their rank range is between 1 and 2. We give 2 as those statements rank value.

B. *Results and Analysis*

We represent the results about the effectiveness with the Siemens subjects in Table 2 and Fig. 4. Table 2 shows the percentage of test runs at each score level. Following the convention of previous studies, [4], [3], [14], each segment is 10 percentage points, apart from the 90-99% range and the 99-100% range because 100% is impossible. "CT" is the Cause-Transitions technique using the standard SDG-ranking technique. For example, there are 25.08% of faulty versions that could be found the bugs easily for less than 1% code to be examined with our technique. The level of 90-99% shows

49.93% of faulty versions only need to be examined less than 10% code.

TABLE I. PERCENTAGE OF TEST RUNS AT EACH SCORE LEVEL

Score	fpFL	LOUPE	Tarantula	SOBER	CT	Cp
99-100%	22.13	19.40	13.93	8.46	10.66	17.74
90-99%	50.82	54.00	41.80	43.84	24.59	27.42
80-90%	18.03	12.90	5.74	21.54	65.57	25.81
70-80%	2.46	5.30	9.84	3.85	74.59	4.84
60-70%	2.46	4.8	8.20	4.62	81.97	4.84
50-60%	2.46	1.1	7.38	0.77	97.54	8.06
40-50%	0.82	0.01	0.82	0.77	100	2.42
30-40%	0.82	2.49	0.82	2.31	100	5.65
20-30%	0	0.00	4.10	2.31	100	2.42
10-20%	0	0.00	7.38	2.31	100	0.81
0-10%	0	0.00	0.00	9.23	100	0.0

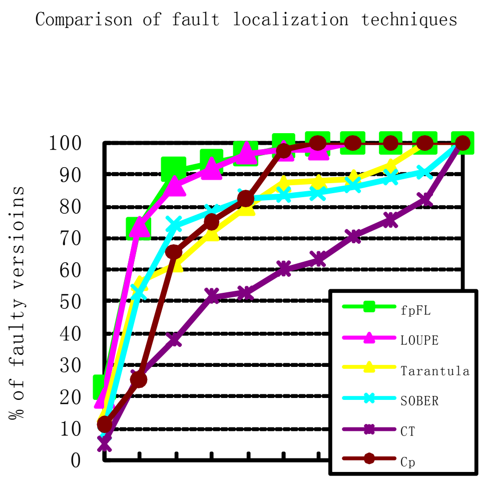


FIGURE IV. COMPARISON OF THE EFFECTIVENESS OF EACH TECHNIQUE

Figure 4 shows a graphical view of the data in Table 2. The horizontal axis represents the lower bound of each score range, and the vertical axis represents the percentage of faulty versions achieving a score greater than or equal to that lower bound. Points and connecting lines are drawn for each technique. For example, the curve of our technique shows that about 22.13% faulty versions have a percentage of between 99%-100%, which means only less than 1% code need to be examined before finding out the bug.

From the figure we can see the advantages of the FPFL algorithm intuitively. In the figure of the statistics, only in the 90 score FPFL achieved a 72.95% version of the rate, less than 73.90% of LOUPE, and the rest of the points curve have achieved the highest value. For the programmer to locate the fault, the less amount of code need to be checked, the greater significance of fault location technology is. It can also be seen from Figure 4, the FPFL algorithm has more advantages than other methods, which can find fault in the program more effectively.

V. CONCLUSION AND FURTHER WORK

In this paper, we present an improved method of software fault localization based on fault propagation. We use the Siemens suite to validate the effectiveness of our method, and most of the faulty version in Siemens suite contains a single fault. Experimental results show that the proposed method has achieved good efficiency in fault localization and enhances the effectiveness for fault localization, which achieved better than LOUPE, Cp etc. The experimental results show the technique can be effective in assisting a programmer locate faults in many cases. However, our technique still has space to improve. So we would like to disinter more information and adjust our model to improve effectiveness in the future. For the future work we plan to extend our framework with run-time program dependences, and use more real-life software to detect our technique. Also we will do more experiments with multi-bugs programs in the near future.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 61672122, No. 61402070, No. 61602077), the Natural Science Foundation of Liaoning Province of China (No. 2015020023), the Educational Commission of Liaoning Province of China (No. L2015060) and the Fundamental Research Funds for the Central Universities (NO. 3132016348).

REFERENCES

[1] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, “Lightweight fault-localization using multiple coverage types,” in Proc. IEEE 31st Int. Conf. Software Engineering ICSE 2009, pp. 56–66, 2009.

[2] Shu T, Ye T, Ding Z, et al. Fault localization based on statement frequency[J]. Information Sciences,Vol 360,pp.43–56 , September 2016.

[3] K. Yu, M. Lin, Q. Gao, H. Zhang, and X. Zhang, “Locating faults using multiple spectra-specific models,” pp. 1404–1410, Mar 2011.

[4] G. K. Baah, A. Podgurski, and M. J. Harrold, “The probabilistic program dependence graph and its application to fault diagnosis,” vol. 36, no. 4, pp. 528–545, 2010.

[5] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, “Regression test selection for java software,” SIGPLAN Not., vol. 36, pp. 312–326, October 2001.

[6] M. Weiser, “Programmers use slices when debugging,” Communications of the ACM, pp. 446–452, 1982.

[7] A. Orso, S. Sinha,, “Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging,” ACM Trans. Softw. Eng. Methodol., vol. 13, pp. 199–239, April 2004.

[8] A. Podgurski and L. A. Clarke, “A formal model of program dependences and its implications for software testing, debugging, and maintenance,” vol. 16, no. 9, pp. 965–979, 1990.

[9] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, “On the accuracy of spectrum-based fault localization,” in Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp. 89–98, IEEE Computer Society, 2007. Ochiai.

[10] M. Hutchins, H. Foster, T. Goradia, “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria,” in Proceedings of the 16th international conference on Software engineering, ICSE ’94, pp. 191–200, IEEE Computer Society Press, 1994.

[11] D. Jeffrey, N. Gupta, and R. Gupta, “Fault localization using value replacement,” in ISSTA ’08: Proceedings of the 2008 international symposium on Software testing and analysis, (New York, NY, USA), pp. 167–178, ACM, 2008.

[12] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, “Capturing propagation of infected program states,” in Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC/FSE ’09, pp. 43–52, ACM, 2009.

[13] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries,” in ASE, pp. 30–39, IEEE Computer Society, 2003.

[14] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, “Sober: statistical model-based bug localization,” SIGSOFT Softw. Eng. Notes, vol. 30, pp. 286–295, September 2005.

[15] Sun, Xiaobing, et al. “IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra,” in Frontiers of Computer Science, vol. 10, pp. 812–831,January 2016.