SEEDED FAULTS AND THEIR LOCATIONS DESIGN USING BAYES FORMULA AND PROGRAM LOGIC IN SOFTWARE TESTING

Wang Lina
Beijing Key Laboratory of Digital Media, School of Computer Science and Engineering
Beihang University, Beijing, China
National Key Laboratory of Science and Technology on Aerospace Intelligent Control
Beijing Aerospace Automatic Control Institute, Beijing, China

Tian Jie
Beijing Command College of CPAPF, Beijing, China

Li Bo
Beijing Key Laboratory of Digital Media, School of Computer Science and Engineering
Beihang University, Beijing, China

Keywords: Software testing, Fault seeding, Procedural language, Fault classification.

Abstract: Focusing on three questions “what faults to seed”, “how to seed faults more effectively” and “how to select the seeded fault locations”, the methods of fault seeding are studied. Aiming at procedural language source code, a fault classification scheme is presented. Referring to Howden’s fault classification scheme, and based on the occurrence causes and manifestations of software faults, a hierarchy of fault classes is designed. The faults are categorized as assignment faults, control flow faults or runtime environment faults. Then they are further classified by degrees, respectively. 96 categories are included in all. According to this classification, a statistical method based on Bayes formula is designed to determine the manifestations of seeded faults. A logical method based on the logical relation between control flow and data flow of program is presented to set seeded locations. And the concrete seeding process is introduced. Finally, the methods are verified by a case.

1 INTRODUCTION

In order to resolve the creditability problem of software testing results, the ability of the testing strategy should be evaluated firstly. Fault detectability and detection effectiveness are effective measures to evaluate testing techniques (Basili and Selby, 1987; Shen et al., 1985). So we can evaluate the effectiveness of the testing strategy according to the ability to detect certain types of faults which are seeded into programs in terms of analysis and practical experiences. Software fault seeding refers to introducing software faults into programs, usually to estimate the number of faults, measure test effectiveness and reliability, evaluate test detectability, or compare testing strategies (Stephens, 2001; Boehm and Port, 2002; Copeland and Haemer, 2000; Meek and Siu, 1989; Offutt and Hayes, 1995; Scott and Wohlin, 2008; Pocatilu, 2010).

It has been proved by practice that if the same testing strategies are applied to different types of programs, the testing results are different. A testing strategy that is effective for all faults in an arbitrary program is inexistent (Girgis and Woodward, 1986; Selby, 1986). Classifying faults on some principles firstly and then de signing testing methods in terms of the classification can improve test effectiveness. The occurrence probability of each type of fault is unequal to another. After classifying faults, different

Lina W., Jie T. and Bo L..
SEEDED FAULTS AND THEIR LOCATIONS DESIGN USING BAYES FORMULA AND PROGRAM LOGIC IN SOFTWARE TESTING.
DOI: 10.5220/0003413002030210
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 203-210
ISBN: 978-989-8425-57-7
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

203

measures can be taken to faults with unequal occurrence probability. This can increase test efficiency. Referring to Howden's fault classification (Howden, 1976) and based on the different occurrence causes, manifestations and characteristics of faults, a fault classification scheme which aims at procedural language source code is put forward. The classification scheme lays one of the foundations for the designment of seeded faults.

The concept of fault seeding has been put forward since at least the early 1970s', but it has not been widely used because of many difficulties (Stephens, 2001; Boehm and Port, 2002; Copeland and Haemer, 2000; Meek and Siu, 1989). One of the most important reasons is that there are few available strong alternative approaches. Therefore, many scholars consider it is worth exploring new approaches to fault seeding, which can be used in practice.

The key issue of fault seeding is that it requires a way to introduce "representative" faults (Stephens, 2001; Boehm and Port, 2002; Meek and Siu, 1989; Offutt and Hayes, 1995). This paper studies on this issue. Focusing on three questions "what faults to seed", "how to seed faults more effectively" and "how to select the seeded fault locations", large numbers of work have been done, and a set of effective methods which apply to procedural language programs have been presented. On the basis of the fault classification scheme, a statistical method based on Bayes formula is designed to select the seeded "content". Based on the logical relation between control flow and data flow of a program, a method used to select seeded fault locations is designed. Then the problem about seeded fault locations is solved. The context of the above work is procedural language programs.

In section 4, the above methods are applied to a real program. The size of the program is 4781 LOC. And sixteen faults were seeded into the program. Then the faulty version of the program was tested. The results of the experimentation prove that these methods are effective and feasible.

2 FAULT CLASSIFICATION

For different purposes, some different software fault classification schemes have been put forward (Kuhn, 1999; Zeil, 1989; Harrold and Offutt, 1994; Telles and Yuan, 2001). Starting with the occurrence causes of software faults in procedural language programs, a fault classification scheme which aims at source code is put forward. This classification lays the foundation for solving the question of "what faults to seed".

2.1 The Occurrence Causes of Faults

The occurrence cause of software faults in procedural language programs includes the following three aspects.

Firstly, the assignment statement of procedural language program disobeys the Referential Transparency principle which is a basic axiom of mathematical deduction. This brings about a series of side effects.

Secondly, in procedural language programs, the change of program execution order is owing to conditional transfer or unconditional transfer. Condition and decision are the premises of conditional transfer. Producing condition correlates with assignment statement nearly. Decision requires to execute various test instructions or comparison instructions. Dijkstra and his supporters claim that unconditional transfer is one of the important reasons which cause programs to make faults (Clark, 1984). Some scholars consider the number of unconditional transfer instructions as a predictor of faults in a program (Shen et al., 1985). Consequently, control and decision are important reasons that lead to various faults in procedural language programs.

Thirdly, some software faults about designing or coding could create circumstances under which will cause computing environment faults. And the program execution must depend on computing environment. So this will cause failed execution of the program.

The above three fundamental reasons are important theoretical bases of the fault classification scheme.

2.2 Fault Manifestations

By analyzing the manifestations of real faults which were detected in tests, the fault manifestations in procedural language programs are divided into three categories.

Firstly, when faults occur, calculation in the program is incorrect. This produces incorrect outputs, but the control flow cannot be changed.

Secondly, when faults occur, the program takes an incorrect path in the execution process, namely, the incorrect jumps or loops are caused. This makes the program control flow changed.

Thirdly, when faults occur, circumstances which cause runtime environment incorrect are generated, so the program cannot execute successfully.

2.3 Fault Classification Scheme

According to the occurrence causes of faults and real faults which were detected in tests, concerning fault manifestations, domain/computation fault classification scheme developed by Howden (1976) is enhanced. Software faults can be categorized as assignment faults, control flow faults or runtime environment faults, which are given below. This classification scheme applies to procedural language programs.

An assignment fault refers to that the fault only causes calculation outputs incorrect, but cannot affect the execution path. Assignment faults only affect the data flow, but cannot affect the control flow. Assignment faults are further classified into two categories. An assignment statement fault occurs during calculation. A procedure call and execution fault occurs during calling, returning or executing the called procedure. Then according to the concrete occurrence reasons, assignment statement faults and procedure call and execution faults are further classified by degrees, respectively. Thirty-three categories are included in all.

A control flow fault refers to the fault which causes a program to take the incorrect path. Control flow faults are further classified into two categories. A decision fault occurs when a program takes the incorrect path at the decision point, or the required path is missing because of the incomplete conditions. A control fault occurs when the control flow of a program is incorrect because of other reasons, although the predicate of the program is correct. Then according to the concrete occurrence reasons, decision faults and control faults are further classified by degrees, respectively. Fifty-one categories are included in all.

The runtime environment cannot generate faults of oneself under normal circumstances. But sometimes the conditions that make runtime environment abnormal are created because of the fault in a program. This fault is called runtime environment fault. And runtime environment faults are categorized as memory faults and interrupt faults. Then they are further classified by degrees, respectively, according to the concrete occurrence reasons. Twelve categories are included in all.

Then a hierarchy of fault classes is designed. Figure 1 only shows the first, the second and the third levels because of the limit of the paper length.

3 FAULT SEEDING METHODS

This section introduces the methods of designing seeded fault manifestations and determining seeded locations. These methods provide effective solutions for the problem about "what faults to seed", "how to seed faults more effectively" and "how to select the seeded fault locations".

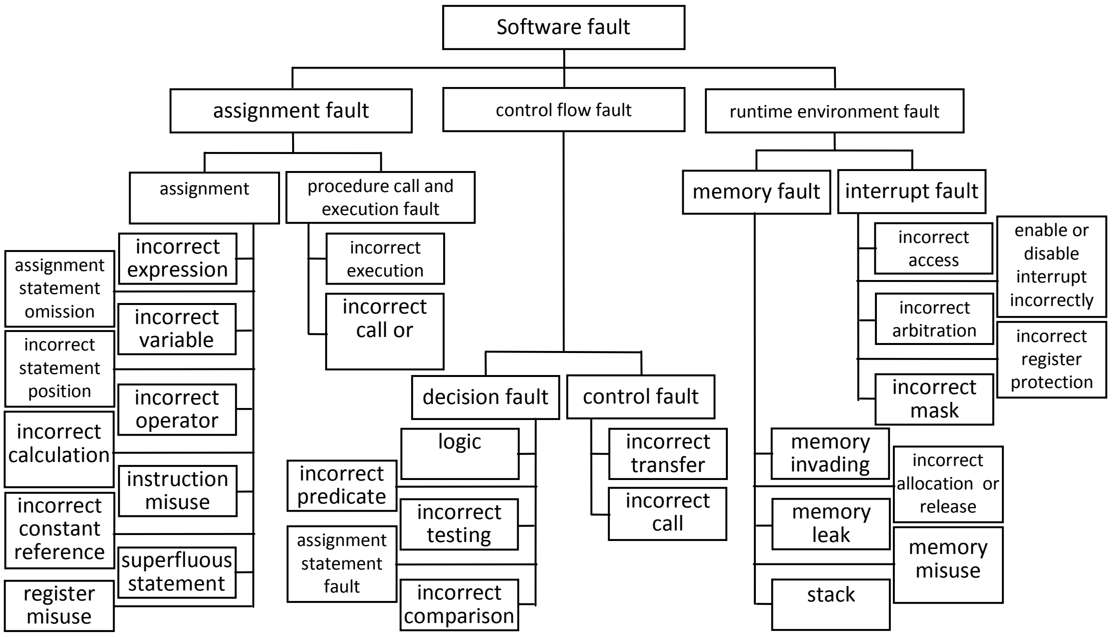

Figure 1: The above three levers of the fault classes.

3.1 The Application of Bayes Formula to Fault Seeding

Practice has proved that each type of fault includes many manifestations. Bayes formula is used to compute their occurrence probability, respectively.

Suppose that F_i belongs to the fault type i of the fault classification and E_{ij} belongs to the manifestation type j of the fault type i, then the expressions are as follows:

$P(F_i)$ is the occurrence probability of F_i,

$P(E_{ij})$ is the occurrence probability of E_{ij},

$P(F_i|E_{ij})$ is the probability that the manifestation type j belongs to fault type i,

$P(E_{ij}|F_i)$ is the probability that the manifestation of fault type i is type j.

Then according to Bayes formula, $P(E_{ij}|F_i)$ is calculated:

$$P(E_{ij} \mid F_i) = \frac{P(E_{ij})P(F_i \mid E_{ij})}{\sum_{q=1}^{m} P(E_{iq})P(F_i \mid E_{iq})} \qquad (1)$$

$(i=1, 2, 3 \ldots n, j=1, 2, 3 \ldots m)$.

The real faults were collected from many programs in former tests. And classify them according to the above fault classification scheme. Then $P(F_i)$, $P(E_{ij})$ and $P(F_i|E_{ij})$ can be calculated. For example, suppose that one hundred real faults can be collected, and three faults therein belong to type i, then $P(F_i)$ can be calculated: $P(F_i) = 3\%$.

In terms of the above formula, $P(E_{ij}|F_i)$ can be easily calculated. It refers to the occurrence probability of each type of fault manifestation E_{i1}, $E_{i2} \ldots E_{im}$, when a type of fault F_i occurs.

From these results, it can be concluded that which fault manifestations of a type occur more frequently in programs. During fault seeding, these fault manifestations should be designed after the type of fault is decided. The advantage of this method is that it can make the seeded faults "representative" and the most effective.

3.2 Seeded Locations

A procedural language program is composed of control flow and data flow. Figure 2 is a sketch map of the program structure. The node "start" stands for the start of a program. The solid arrow "c" stands for the control flow. And the solid arrow "d" stands for the data flow. If the control flow and the data flow correlate, the solid arrow "r" stands for the action which a program goes into a loop, and the solid arrow "j" stands for the action which the control flow transfers under the influence of the data flow.

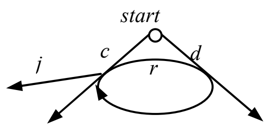

Figure 2: Program structure hint.

According to Figure 2, the relation between control decision and assignment of a program can be deduced, as Figure 3 shows. This figure includes nodes "IF" which stand for control and decision of a program, nodes ":=" which stand for assignment, and solid arrows which stand for the control flow or the data flow of a program. Figure 3 indicates the correlation phenomenon between control decision nodes and assignment nodes. Among the congeneric nodes, the upper level nodes affect the lower nodes, and this is also indicated in Figure 3. If a program is analyzed and the seeded locations are determined in terms of Figure 3, the conditions are too complex to deduce the conclusions. So this figure must be decomposed and simplified.

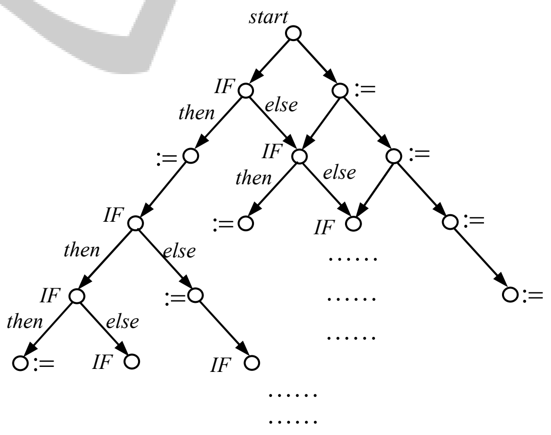

Figure 3: The correlation between control and decision nodes and assignment nodes of a program.

During fault seeding, a simplified analysis method is adopted. In this method, the control decision and assignment of a program are analyzed separately. Figure 3 is divided into the control and decision sketch map, such as Figure 4, and the assignment sketch map, such as Figure 5. Analyzing the control decision and the assignment of a program in terms of the practical requirements and applications, respectively, can make the problem easy, and make the process clear. Consequently, this approach facilitates practical applications. Practice has proved that this simplification is feasible.

Figure 4 is an asymmetric binary tree. Each node "IF" has two branches. They correspond to the optional paths "then" or "else" and point to a new node "IF", respectively. This figure shows a simple hint of the control and decision process of a program. Figure 5 is a hierarchy chart. There are some assignment nodes on each level, and the number of nodes increases level by level. The assignments of the lower level nodes are influenced by the upper nodes. This figure shows a simple hint of the assignment process of a program.

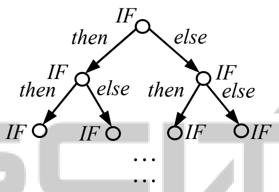

Figure 4: Control and decision hint of a program.

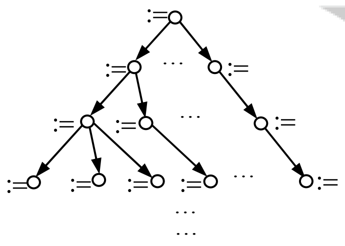

Figure 5: Assignment hint of a program.

The structures, functions and input domains of a program module are analyzed, and the real running circumstantialities of the program in the past are considered thoroughly. Then each branch of a node "*IF*", that is the path "*then*" or "*else*" of Figure 4, is assigned a weight based on the occurrence probability. Afterwards, the weight of each node "*IF*" can be obtained by calculating.

The following illustrates the concrete applications of the above method. Figure 6 is a flow chart of a program module. The rhombuses stand for the control and decision node. Obviously, the first to the fifth control and decision nodes of Figure 6 correspond to nodes "*IF*" of Figure 4. Suppose that *IF*$_1$, *IF*$_2$, *IF*$_3$, *IF*$_4$ *and IF*$_5$ represent the control and decision nodes of Figure 6, respectively. Their two branches "Y" and "N" correspond to the optional paths "*then*" or "*else*" of Figure 4, respectively. Figure 6 also gives the weight coefficients of all branches, such as ω_1, $\omega_1^{'}$, and so on. Figure 7 is a control and decision sketch map with weights of the program module.

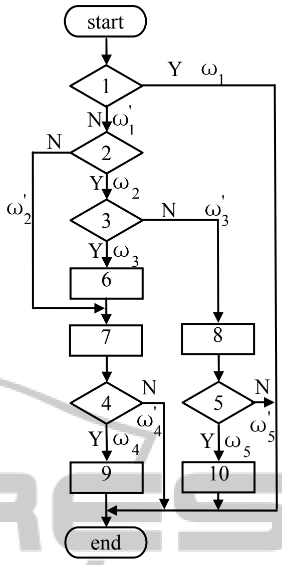

Figure 6: The flow chart of the program module.

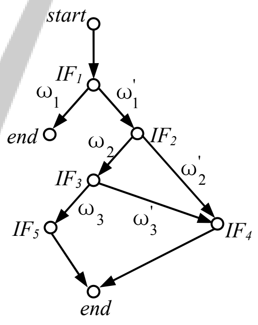

Figure 7: Control and decision hint of the module with weights.

Linking theoretical analysis with real running circumstantialities, the occurrence probabilities of "Y" and "N" branches of each control and decision node are calculated. Then ω_i, $\omega_i^{'}$ (*i*=1, 2, 3, 4, 5) are assigned as follows:

$$\omega_1 = 0.1, \ \omega_1^{'} = 0.9 \quad (2)$$

$$\omega_2 = 0.7, \ \omega_2^{'} = 0.3 \quad (3)$$

$$\omega_3 = 0.55, \ \omega_3^{'} = 0.45 \quad (4)$$

$$\omega_4 = 0.6, \ \omega_4^{'} = 0.4 \quad (5)$$

$$\omega_5 = 0.8, \ \omega_5^{'} = 0.2 \quad (6)$$

Then the weights of all nodes "*IF*" of the program module are given below:

$$\text{The weight of } IF_1 = 1 \quad (7)$$

$$\text{The weight of } IF_2 = \text{the weight of } IF_1 * \omega_1' = 0.9 \quad (8)$$

$$\text{The weight of } IF_4 = (\text{the weight of } IF_2 * \omega_2') + (\text{the weight of } IF_3 * \omega_3) = 0.62 \quad (9)$$

$$\text{The weight of } IF_5 = \text{the weight of } IF_3 * \omega_3' = 0.28 \quad (10)$$

Then the weights of all nodes "*IF*" of the program can be figured out. That is to say, the weights of all control and decision nodes of the program can be obtained. During fault seeding, the control and decision nodes with less weight should be selected to be optional seeded locations, because these parts are more difficult to reach and this can make the influence of the seeded faults on the program smaller. Consequently, this can force the test cases to be of higher efficiency. Thus the problem of "how to select the seeded fault locations" is solved.

4 FAULT SEEDING AND TEST CASE

This section introduces concrete steps of fault seeding using the above methods by a case. Some artificial faults were seeded into a subject assembly program. And the faulty versions of the program were tested.

4.1 Historical Statistics of Real Faults

Collecting the detected faults from many programs during recent years, and analyzing their occurrence reasons, the type of each fault can be determined. Table 1 shows the proportion of each type of fault.

In this study, these programs are real-time embedded software and written in C or assembly language. Aiming at different types of programs or different organizations' programs, the different experiences and historical data of real faults should be collected. That is to say, their own data as table 1 should be presented. This can lay the foundation for "representative" fault designing.

4.2 Case Study

This paper seeds faults by the following five steps. The subject program is written in assembly language. And the size of the program is 4781 LOC.

In the first step, the source code of the subject program was entirely analyzed.

Table 1: Proportion of Each Type of Fault.

Type of Fault	Subcategory	Percentage of each Subcategory	Percentage of each Type
Assignment faults	Procedure call and execution faults	1.8%	30.9%
	Assignment statement faults	29.1%	
Control flow faults	Decision faults	45.5%	48.2%
	Control faults	2.7%	
Runtime environment faults	Interrupt faults	16.4%	20.9%
	Memory faults	4.5%	

All computing nodes of the program were found out. Their computing results were analyzed, respectively, to determine whether they had effects on decisions of path selecting in the program. The number of all conditional transfer instructions and all unconditional transfer instructions were counted, respectively. Table 2 shows the statistical results.

Table 2: Percentage of Various Transfer Instructions in the Subject Program.

Instruction	Number	Percentage	Instruction	Number	Percentage
JMP	77	37.6%	JZ	47	22.9%
JA	13	6.3%	JNC	2	1%
JAE	6	2.9%	JNE	4	1.95%
JB	20	9.8%	JNZ	31	15.1%
JE	3	1.5%	JNA	2	1%
JP	3	1.5%	Sum	205	100%

In the second step, the types and the manifestations of the seeded faults were initially designed.

From the work introduced in subsection 4.1, some results can be deduced. The percentage of assignment faults is 30.9%. Some incorrect changes of the control flow were caused by assignment statement faults, and the percentage is 62.9%. Consequently, during selecting the types of the seeded faults, assignment statement faults were considered emphatically. From the above classification, eight assignment faults were selected. Table 3 shows the details about the types of these faults.

The subcategories of seeded control flow faults were decided in terms of the data of table 1. The transfer instructions which occurred more frequently were selected as subject transfer instructions in terms of the data of table 2. Six control flow faults were selected from the above classification. Table 3 shows the details about the types of these faults.

Considering the characteristics of the subject program, two runtime environment faults were selected. And they were an interrupt fault and a memory fault.

By using the methods presented in subsection 3.1, the manifestations of seeded faults were designed.

In the third step, seeded locations were determined, and the artificial faults were seeded into the subject program one by one.

In the source code of the subject program, for each artificial fault which was designed in the second step, the code nodes which can be regarded as seeded locations were found out and marked. Assignment faults should be located on the computing nodes and these nodes must have no influence on control flow of the program. Control flow faults should be located on the control and decision nodes which include conditional transfer instructions, unconditional transfer instructions or call instructions. Some computing nodes of the program affect the control flow change. Control flow faults can also be located on these nodes. Runtime environment faults should be located on the code nodes which have effects on software runtime environment.

According to the methods presented in subsection 3.2, the weights of all marked control and decision nodes were calculated. The nodes with less weight were considered as alternate seeded locations. Considering the characteristics of the program language, the structures and functions of the subject program and so on, the seeded locations for each fault were initially selected. Then the faults were seeded into the subject program one by one.

After seeding a fault, the faulty version of the subject program was compiled to build an executable file. If it is successful, another fault could be seeded. If it is unsuccessful, the seeded location was modified until generating executable file successfully.

In the fourth step, the faulty version of the subject program was tested in order to improve seeding.

Under the current testing conditions, the faulty version of the subject program was tested using the real testing strategies. If some seeded faults were masked by others, or the output data fell into confusion, or some seeded faults correlated with others etc., the testing strategies can not be evaluated effectively. Then the third step needs to be repeated for the purpose of improving the seeded locations.

In the fifth step, the fourth step was repeated until all artificial faults which were designed in the second step were seeded into the subject program.

4.3 Tests

The faulty version of the subject program was tested by validation testing. Each validation test case was conduced. And table 3 shows the test results. Three faults were not detected. They are a variable initial value fault, a computational accuracy fault and a closing interrupt omission fault. The variable initial value fault belongs to variable faults. The computational accuracy fault belongs to calculation faults.

Table 3: Fault Seeding in the Subject Program and Testing Results.

Type of Fault	Subcategory of Fault	Number of seeded faults	Number of detected faults
Assignment faults	Incorrect parameter passing	1	1
	Incorrect variable	2	1
	Incorrect statement position	1	1
	Incorrect constant reference	1	1
	Incorrect calculation	2	1
	Register misuse	1	1
Control flow faults	Incorrect testing	1	1
	Incorrect comparison	1	1
	Incorrect boolean	1	1
	Condition omission	1	1
	Transfer instruction misuse	1	1
	Call omission	1	1
Runtime environment faults	Omission of disable interrupt	1	0
	Memory invading	1	1

This variable initial value fault is a maskable fault. The initial value of the variable was assigned in the variable definition section, but it was not used in the program. Consequently, no matter what the initial value was, there was no influence on the program. This brought hidden trouble to the program reliability. As for the computational accuracy fault, the data type of the variable was changed from "DQ" to "DD". Although the accuracy was reduced, the influence was small because the value of the variable was large. So the change was not detected by comparing testing data. After seeding the closing interrupt omission fault, there was no other interrupts during the program running. So it did not cause any influence on the program. This also could bring hidden trouble to the program security and reliability.

5 CONCLUSIONS

This paper presents a source code oriented fault classification scheme. The classification scheme was applied to software fault seeding. Regarding the essence of the procedural language and the occurrence causes of software faults as theoretical foundations, and considering three aspects about assignment statements, control decision and runtime environment, the software faults are classified as assignment faults, control flow faults or runtime environment faults. Then they are further classified by degrees, respectively, according to the concrete occurrence reasons. That is to say, a hierarchy of fault classes is designed.

A statistical method based on Bayes formula is presented, which can provide a guarantee for "representative" faults seeding. A logical method based on the logical relation between control flow and data flow of a program is also presented, which can be used to determine the seeded locations rationally. After seeding faults into the subject program, and testing the faulty version, the fault detectability and detection effectiveness can be measured by analyzing testing results. This can provide important hints for the testing strategy improving.

In the future work, large numbers of naturally occurring faults will be collected, and this will further improve the authenticity of statistical data. Moreover, this can make the types and the manifestations of seeded faults which are designed during fault seeding more "representative". The proposed methods lay theoretical foundations for fault seeding. According to the requirements of the real project, the methods can be adjusted properly, and the automatic fault seeder will be designed.

ACKNOWLEDGEMENTS

I am grateful to Zhang Fan, Dong Zhanqiu and Hu Yubiao for their suggestions.

REFERENCES

Basili, V. R., Selby, R. W., 1987. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, vol. 13, no. 12, December.

Boehm, B., Port, D., 2002. Defect and Fault Seeding In Dependability Benchmarking. *DSN Workshop on Dependability Benchmarking*.

Clark, R. L., 1984. A Linguistic Contribution to Goto-Less Programming. *Communications of the ACM*, vol. 27, no. 4, April.

Copeland, J., Haemer, J. S., 2000. The Art of Software Testing. *SW Expert*.

Girgis, M. R., Woodward M. R., 1986. An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria. In *Proc. Workshop on Software Testing*. IEEE Computer Society Press.

Harrold, M. J., Offutt, A. J., Tewary, K., 1994. An Approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph. In *Proceedings of International Symposium on Software Reliability*

Howden, W. E., 1976. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, vol. 2, no. 3, September.

Kuhn, D. R., 1999. Fault Classes and Error Detection Capability of Specification-Based Testing. *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 4, October.

Meek, B., Siu K., 1989. The effectiveness of error seeding. *ACM Sigplan Notices*, vol. 24, no. 6, June.

Offutt, J., Hayes, J. H., 1995. A Semantic Model of Program Faults. *ISSE-TR-95-110*.

Pocatilu, P., 2010. Quality Related Costs of e-Business Systems. *Journal of Applied Collaborative Systems*, vol. 2, no. 2.

Scott, H., Wohlin, C., 2008. Capture-recapture in Software Unit Testing: A Case Study. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*.

Selby, R. W., 1986. Combining Software Testing strategies: An Empirical Evaluation. In *Proc. Workshop on Software Testing*. IEEE Computer Society Press.

Shen, V. Y., Yu, T. J., Thebaut, S. M., Paulsen, L. R., 1985. Identifying Error-prone Software--An Empirical Study. *IEEE Transactions on Software Engineering*, vol. 11, no. 4, April.

Stephens, R. T., 2001. Dynamic Duo: Code Coverage and Fault Seeding. *System Development Process*.

Telles, M., Yuan, H., 2001. The Science of Debugging. *The Coriolis Group LLC, 14455N.*

Zeil, S. J., 1989. Perturbation Techniques for Detecting Domain Errors. *IEEE Transactions on Software Engineering*, vol. 15, no. 6, June.