Program Segment Testing for Software Fault Prevention

Lei Rao[1, a)] Shaoying Liu[1, a)]
Ai Liu[1]

Abstract: Fault prevention is a process of quality improvement which aims to identify common causes of faults and take relevant measures to prevent the type of fault recurrence. However, not only is there little related research at present, but almost no work to automatically and effectively improve the code to prevent potential faults from being introduced during the programming process for the root cause of the fault. In this paper, we propose an automatic fault prevention technology called *Program Segment Testing* (PST), which can automatically prompt whether a certain piece of code will trigger an exception during the programming process. First, we introduce some preliminary definitions and notation that are used in our technology. Then, we explain PST in detail, and point out the problems that need to be solved to complete this technology and the corresponding solutions. Finally, we give a summary to our work and point out that our method provides a way to prevent the trigger from the root cause of the exception.

Keywords: Fault prevention, Exception, Segmenting criterion, Program segment Testing

1. Introduction

Software fault prevention is an important activity in the development cycle of any software project. Starting from the early stage of the project, this measure is appropriate in order to prevent faults from being introduced into the product. Therefore, it improves the quality of a software products and reduces the cost of subsequent maintenance.

However, according to the literature, most software project teams focus on fault removal [1, 2]which through testing or debugging to detect and remove the faults in the program after the program is completed; or focus on fault tolerance[3, 4]which using redundancy to ensure that the program keep work normally when a fault occurs. Their attention is paid to how to solve the faults after the program is completed. Few people think about taking some measures directly in the process of programming to prevent faults from being introduced into the final project. Moreover, in the few related works, techniques such as mining frequent itemsets [5, 6]and detecting risky commits[7]are mainly used to use the correlation between faults for rough prevention. There is a high probability that misjudgments or omissions will occur, because the introduction of faults is not prevented from the root cause.

In order to solve the above-mentioned problems, based on the idea of Human-Machine Pair Programming proposed by Prof.Liu recently[8], we propose an automatic fault prevention technology called *Program Segment Testing* (PST), which can automatically prompt whether a certain piece of code will trigger an exception during the programming process. Specially, our method includes following functions: (1) Automatically intercept a segment that may trigger an exception from the program being programmed according to the segmenting criterion, and wait for subsequent testing; (2) Automatically identify all variables in this segment, and initialize the variables except Key Variables(KV), which will be explained in the second section, and combine with the segment to create a new executable test program; (3) Automatically generate test values for KV, and executes the test program to get a result; (4) If the result does not match the expectation, it will automatically feed back to the programmer, prompting that there is a problem in this code.

The reminder of the paper is outlined as follows. Section 2 introduces some preliminary definition and notion that are used in our method. Section 3 presents the method we propose in detail. Section4 compares our method with related work. Section5 concludes the paper and points out future research direction.

2. Preliminary and Notation

This section introduces some preliminary definitions and notation that are used along the paper.

Program segmenting is based on a segmenting criterion over which the segment is obtained. In traditional program slicing, the criterion usually corresponds to a statement in the code and a variable within that statement[9]. However, if we use statements in our method, it will not be as precise as we want to be. Therefore, we base our segmenting criterion on expressions, which do not impose that precision barrier.

Definition1(Segmenting Criterion). Let P be a program. A segmenting criterion C_E of P is a key expression that determines whether the exception E is triggered or not.

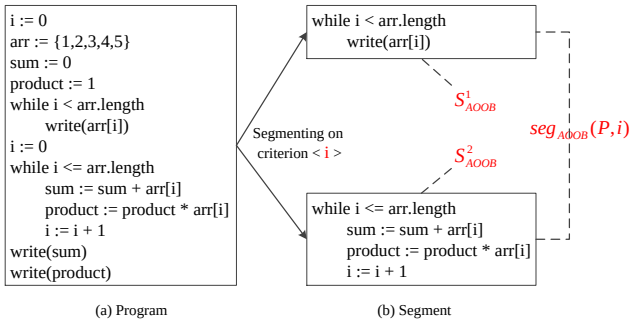

Fig.1 Segmenting Criterion

Therefore, in order to be able to intercept the segment we want more accurately, we define the segmenting criterion as an expression that can determine whether an exception is triggered. Moreover, a variable can be considered as a segmenting criterion in our definition because variables are expressions.

1 Hiroshima University, Higashi-Hiroshima, Hiroshima 739-0036, Japan

For example, an array index overflow error often occurs in the program, and the reason for the error is an attempt to call an element outside the array index range, so we use the array index *i* as a segmenting criterion to intercept the program as shown in Fig.1(a).

Definition2(Segment). Let P be a program, E be an exception and C_E be a segmenting criterion for E. $\mathrm{seg_E(P, C_E)}$ is the finite set containing all segments obtained by intercepting program P using the segmenting criterion C_E for the exception E, where each segment is denoted as S_E^i, i is from 1 to n, n is the number of all segments for exception E and E is the type of exception.

After intercepting the program with the index variable *i*, we get the segments shown in Fig.1(b), namely

$$seg_{AOOB}(P, C_{AOOB}) = seg_{AOOB}(P, i) = \{S^1_{AOOB}, S^2_{AOOB}\}$$

where *AOOB* stands for the common exception *ArrayOutOfBounds*. As for why the intercepted segment is like this, we will give an explanation in the follow-up content of the paper. In addition, it can be seen from the definition that segment S_E^i contains the following attributes:

(1) S_E^i can be obtained by intercepting code from P, denoted as $S_E^i \subseteq P$.

(2) The sum of all S_E^i is equal to $seg_E(P, C_E)$, that is, $\sum_{i=0}^{n} S_E^i = seg_E(P,\ C_E)$

We use V to represent all variables in a segment. Those variables whose wrong value will trigger an exception, we call *Key Variable* (*KV*), and the key variable is included in the segmenting criterion. The remaining variables are called *Related Variable* (*RV*).

In order to determine whether an exception will be triggered in the intercepted segment, it is necessary to take measures such as testing to confirm. The prerequisite for testing a segment is to build the segment into an executable test program. Since for testing, of course, it is necessary to input some test values to these programs. Each type of exception corresponds to a set of test values, which is denoted as I_E. Note that the input $\mathrm{I_E}$ is actually the assignment of *KV*. As we mentioned above, whether a segment will trigger an exception depends on the value of its corresponding *KV*, so we need to give *KV* some suitable test values to observe its results. In addition, an executable program also includes the initialization of the *RV* in the segment, which we denote as R_E^i. Therefore, an executable test program should include a segment S_E^i, input I_E and RV initialization R_E^i.

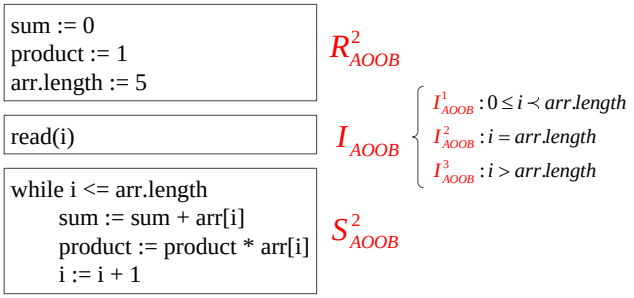

Fig.2 Test Program

Definition3(Test Program). An executable *Test Program* (TP_E^i) includes segment S_E^i, the input I_E, and the assignment of RV, R_E^i, that is

$$TP_E^i = S_E^i + I_E + R_E^i$$

where i is between 1 and n, n is the number of all segments for exception E and E is the type of exception.

We continue the example in Fig.1 to expand S^2_{AOOB} into an executable test program. For the array index overflow problem, the key variable is naturally the index variable i, so we set three test values for it as shown in Fig.2 and the reason will be explained in Chapter 3. In addition to the key variable i, S^2_{AOOB} also contains related variables *sum*, *product* and the length of the array *arr.length*. And the value of these variables, we can use the dependence of the variable to obtain[10, 11]. Thus, we get a test program in Fig.2.

Since judging whether a segment will trigger an exception, it is not only limited to the way of constructing an executable test program, but also includes some other means. Therefore, we can know that the number of all test programs for an exception is less than or equal to the number of segments that it intercepts. The example in Fig.1 happens to be the case where the two are equal.

For each given input, the test program will produce a result. This result is a Boolean variable used to indicate whether the test program will trigger the corresponding exception for the test value. Formally, the definition of execution result is given as follows.

Defination4(Result of Testing): Let P be a program, C_E be a segmenting criterion for the exception E and TP be an executable test program. The execution result of TP_E^i on a segmenting criterion C_E is a Boolean value, including *Correct* and *Incorrect*, which is represented by res (TP_E^i, C_E).

For the example in Fig.2, since the array index i starts from 0, the upper limit of i should be "*arr.length-1*", and we mistakenly wrote "i < *arr.length*" as "*i<=arr.length*", so the test result obtained is Incorrect, which means that this segment will trigger an array index overflow exception.

3. Methodology

In order to ensure that a program is robust, that is, it will not trigger exceptions when the program is running, the most common method is to generate test data to test the completed program under the premise that all possible exceptions listed in the java.lang package are taken into account as much as possible. However, this approach has the following problems: First, a test value usually cannot meet the test requirements of all exceptions at the same time, so we consider generating specific test values for each exception to test the program separately. However, usually there is only a small part of the content related to an exception in a large code, and testing the entire code has problems such as time-consuming and low efficiency. In addition, can we realize that in the process of programming, we can get prompts that a certain piece of code will trigger a certain exception and need to be modified, so as to achieve a "Correct-By-Construction" effect.

In order to solve the above problems, as shown in Fig.3, we propose a *Program Segment Testing* (*PST*) technology that can automatically prompt when a certain piece of code has a problem that will trigger an exception during the programming process.

First, we set a segmenting criterion for each exception respectively. When the programmer writes a piece of code that may trigger an exception, the system will automatically intercept it and wait for subsequent testing. Note that the intercepting we are talking about here is not to directly deduct this code from the program, but to take out this code in a form of copying or mapping with the help of software, and perform subsequent operations on it.

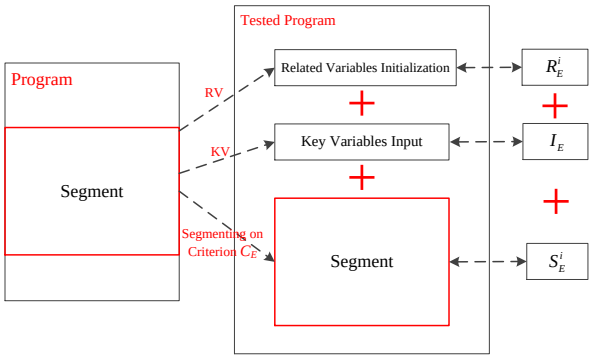

Fig.3 Program Segment Testing

Secondly, for a segment, we use the dependency of variables to obtain the value of the *RVs* closest to the segment, and then add it to the test program in the form of initialization. For *KV*, in order to obtain effective test results, different exceptions correspond to specific test values, that is, the assignment of *KVs*. Therefore, we provide an interface to wait for the input of test data.

Then, we initialize the RV arbitrarily but reasonably, because their values will not affect the test. It is not our focus, but just to form a complete test program. For KV, in order to obtain effective test results, we need to set up specific test values according to different exceptions.

Finally, with the help of the data generation technology[12, 13]to automatically generate the test data we need, and input it into the test program for testing, and the test results will be automatically fed back to the programmer as a basis for judging whether the exception will be triggered.

In summary, in order to complete the entire PST process as shown in Fig.3, the following issues need to be resolved:

(1) What is the interception criterion for each exception? And, how to intercept the appropriate segment from the program?

(2) How to assign values to related variables in the segment?

(3) How to assign values to key variables to achieve effective and efficient test results?

(4) In what form should the test result be fed back to the programmer?

Next, we take a common array index overflow problem as an example to introduce in detail how to solve these problems.

3.1 Segmenting Criterion

Since we are to solve the problem of array index overflow, our focus should be the index of the array, so we choose the commonly used array index *i*, *j* and other variables as the segmenting criterion to intercept the program.

The use of array index in the program can be roughly divided into two types [14]. The first is inside the loop structure. This is the most common and the most prone to array index overflow errors. Therefore, when encountering a loop structure with segmenting criterion as an index variable, we directly intercept the entire loop structure, as shown in Fig.4, and wait for subsequent tests. It should be noted that there are some differences between the for loop and the while loop or the do/while loop. The latter does not include the operation of assigning an initial value to the index variable, so the entire loop structure can be intercepted directly. But in the for loop, the index variable is usually initialized, so when intercepting the for loop, it must be deleted, as shown in Fig.4. Because if we do not do this, even if we give some test value to the index variable, it will be re-assigned every time the loop is executed.

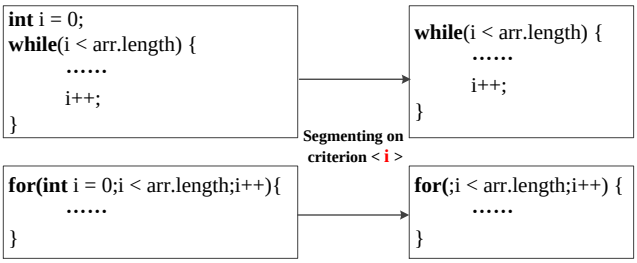

Fig.4 Segmenting on Criterion i

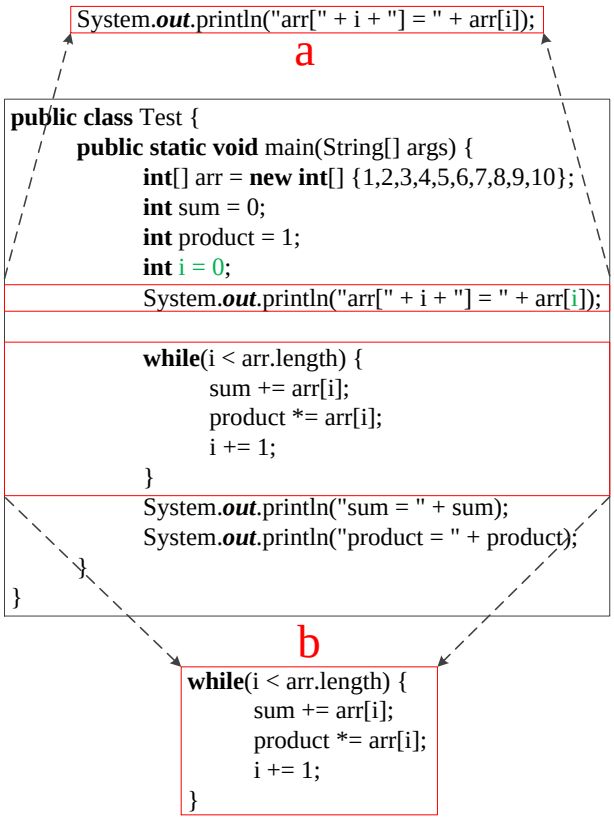

Fig.5 Interception method for array index overflow

The other is outside the loop. When the i-th element in the array is called in a line of code in the program, at this time, this line of code will be intercepted, as shown in Fig.5. This situation is simpler than the previous one. We only need to use the dependency of the variable to retrieve the latest value of the index variable *i* before this line of code, such as the green "*i = 0*" in Fig.5, and then detect the relationship between it and the array length arr.length. If *i* is less than arr.length, the code is correct, otherwise it means that an index overflow exception will be

triggered. Therefore, in the discussion that follows, we mainly focus on the situation where the array index is inside the loop.

3.2 Key Variables

In the case of array index overflow, the key variable is the index variable. Whether a loop structure will trigger the exception of array index overflow depends on whether the loop body statement will be executed when the index is greater than or equal to the length of the array. As shown in Fig.5, under normal circumstances, when the value of i grows to the length of the array arr.length, the loop body statement cannot continue to execute. But if we mistakenly set the range of i to be less than or equal to arr.length in Fig.6, when i is equal to arr.length, the loop body statement will continue to execute, and an exception will be triggered at this time.

```
while(i <= arr.length){
        sum += arr[i];
        product *= arr[i];
        i += 1;
}
```

Fig.6 Wrong value range limit

Therefore, we set three test values for i, which are between 0 and arr.length, equal to arr.length and greater than arr.length, as shown in Table 1. We will explain the reason for setting the test values in this way later.

Table 1 Test value for array index overflow

0<= i < arr.lenth
i = arr.length
i > arr.length

3.3 Related Variables

In a segment, in addition to the key variables, the other variables are related variables, such as sum and product in Fig.5. In fact, on the issue of array index overflow, their values are almost meaningless for testing, because we only care about whether the loop body will execute when the value of the index variable is out of range. But in order to build an executable test program, we need to initialize them. We still use the dependency of variables to get the latest value of sum and product before this segment, as shown in red and green fonts in Fig.7, respectively.

3.4 Testing Result

From Section 3.2, we know that in the array index overflow problem, we can set three test values for the key variable i, and judge whether the segment has an array index overflow vulnerability by detecting whether the loop body statement is executed. The first one is between 0 and *arr.length*. In this case, the loop body can be executed normally, but if it does not, it means that there are other problems in the program, which are not caused by the index value. The second value is that i is equal to *arr.length*. Due to the differences between programming languages, there are two different situations. In a programming language where the array index starts from 0 by default, the loop body cannot be executed if the program is correct. In programming languages where the array index starts from 1 by default, the loop body can still be executed. Since our cases are all written in java language, in this paper we set the index of the array to start from 0. The third value is that i is greater than arr.length. In this case, the loop body statement should not be executed.

```
public class Test {
    public static void main(String[] args) {
        int[] arr = new int[] {1,2,3,4,5,6,7,8,9,10};
        int sum = 0;
        int product = 1;
        int i = 0;
        System.out.println("arr[" + i + "] = " + arr[i]);
        while(i < arr.length) {
            sum += arr[i];
            product *= arr[i];
            i += 1;
        }
        System.out.println("sum = " + sum);
        System.out.println("product = " + product);
    }
}
```

Fig.7 Assignment of related variables

Table 2 Test results for array index overflow

	HasExecute?	IsArrayOutOfBounds?
0 <= i < arr.length	Yes	No
	No	Exist other Faults in Program
i = arr.length	Yes	Yes
	No	No
i > arr.length	Yes	Yes
	No	No

The results obtained by three different test values can reflect different problems, and the basis that can prove that this program will not trigger the exception is that the results obtained by the three test values are *Yes*, *No*, *No*, namely

$$(Res1, Res2, Res3) = (Yes, No, No) \implies Correct$$

Also, the order of the three results cannot be changed. That is to say, as long as the results of the three test values are not *Yes*, *No*, *No* in sequence, the system will automatically feed back to the programmer, prompting that the program has a loophole.

4. Related Work

Association analysis: In programs, there are often some implicit programming rules, that is, two functions are often used together, such as spin lock and spin unlock. Therefore, Zhenmin Li et al. [15] proposed a method to automatically extract implicit programming rules from large software codes, and proposed an effective algorithm to detect violations to the extracted programming rules in the program, that is, find the number of

case that contains the itemset on the left but not those on the right. Yigu Liu et al. mentioned fault association analysis based on numerous historical fault data in [5]. Fault association analysis returns the results in form of {A1, A2}, where A1 and A2 are two different fault types, and {A1, A2} means that fault A1 and A2 frequently occurred in the same period. By discovering knowledge like that, they can implement preventive maintenance for A2 once A1 occurs, and vice versa. However, these fault preventions using the correlation between the elements or faults in the program can only be regarded as a highly reliable method, and cannot accurately diagnose the faults in the program. Moreover, you can only check for possible faults after the program is completed, and cannot prevent errors from being introduced during the programming process.

Fixes form repository: Some program repair tools learn useful fixing strategies by mining bug fixing sets [16], user-debugger interaction [17], human-written patches [18, 19], etc., and then add them to the code analyzer to fix defects in the program. But they cannot determine whether what they are mining is useful and often appears in actual code. In other words, they cannot confirm which errors can be fixed, and cannot guarantee whether the errors that can be fixed are meaningful enough.

5. Conclusion

Based on the idea of Human-Machine Pair Programming, this paper proposes an automatic fault prevention technology *Program Segment Testing* (*PST*), which can automatically test the code during the programming process and prompt that a certain piece of code has a vulnerability that can trigger a certain exception. First, automatically intercept a continuous piece of code that may trigger the exception from the partially completed program according to the segmenting criterion corresponding to an exception, and wait for subsequent testing. Second, automatically identify all variables in this code, initialize RV and combine the intercepted segment to create a new executable test program. Finally, automatically generate some test values for the KV, and execute the test program to get a result. If the result does not match the expectation, it is automatically fed back to the programmer, prompting that this code will trigger the exception.

PST provides a way to prevent faults from the root causes of exception triggers, and can accurately identify which exceptions will be triggered in the code. In addition, it also provides a way to automatically give prompts during the programming process to prevent faults from being introduced into the product. Of course, this technology is still relatively crude, and we will optimize it in more detail in the future.

Reference

[1] X. Zhang, X. Teng, and H. Pham, "Considering fault removal efficiency in software reliability assessment," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans,* vol. 33, pp. 114-120, 2003.

[2] B. Littlewood, "Stochastic reliability-growth: A model for fault-removal in computer-programs and hardware-designs," *IEEE Transactions on Reliability,* vol. 30, pp. 313-320, 1981.

[3] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15,* vol. 10, p. 124, 1985.

[4] B. Randell, "System structure for software fault tolerance," *Ieee transactions on software engineering,* pp. 220-232, 1975.

[5] Y. Liu, S. Gao, and L. Yu, "A novel fault prevention model for metro overhead contact system," *IEEE Access,* vol. 7, pp. 91850-91859, 2019.

[6] K. Gouda and M. J. Zaki, "Efficiently mining maximal frequent itemsets," in *Proceedings 2001 IEEE International Conference on Data Mining*, 2001, pp. 163-170.

[7] M. Nayrolles and A. Hamou-Lhadj, "CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 153-164.

[8] S. Liu, "Software Construction Monitoring and Predicting for Human-Machine Pair Programming," in *International Workshop on Structured Object-Oriented Formal Language and Method*, 2018, pp. 3-20.

[9] M. Weiser, "Program slicing," *IEEE Transactions on software engineering,* pp. 352-357, 1984.

[10] M. Burke and R. Cytron, "Interprocedural dependence analysis and parallelization," *ACM Sigplan Notices,* vol. 21, pp. 162-175, 1986.

[11] K. Muthukumar and M. V. Hermenegildo, "Determination of Variable Dependence Information through Abstract Interpretation," in *NACLP*, 1989, pp. 166-185.

[12] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21-28.

[13] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering,* vol. 16, pp. 870-879, 1990.

[14] B. Chimdyalwar, "Survey of array out of bound access checkers for c code," in *Proceedings of the 5th India Software Engineering Conference*, 2012, pp. 45-48.

[15] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes,* vol. 30, pp. 306-315, 2005.

[16] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 315-324.

[17] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 70-79.

[18] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802-811.

[19] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298-312.

Acknowledgments The research was supported by ROIS NII Open Collaborative Research 2021-(21FS02).