Research on Object-oriented Software Testing Cases of Automatic Generation

Zhang Junli and Song Limin
Sias International College of Zhengzhou University, Henan XinZheng, China
Email: Julyyy123@163.com

Lin Hao Li
AT&T Labs, 200 South Laurel Ave, #D4-3C05, Middletown, NJ, USA
Email: lli@research.att.com

Abstract—In the research on automatic generation of testing cases, there are different execution paths under drivers of different testing cases. The probability of these paths being executed is also different. For paths which are easy to be executed, more redundant testing case tend to be generated; But only fewer testing cases are generated for the control paths which are hard to be executed. Genetic algorithm can be used to instruct the automatic generation of testing cases. For the former paths, it can restrict the generation of these kinds of testing cases. On the contrary, the algorithm will encourage the generation of such testing cases as much as possible. So based on the study on the technology of path-oriented testing case automatic generation, the genetic algorithm is adopted to construct the process of automatic generation. According to the triggering path during the dynamic execution of program, the generated testing cases are separated into different equivalence class. The number of testing case is adjusted dynamically by the fitness corresponding to the paths. The method can create a certain number of testing cases for each execution path to ensure the sufficiency. It also reduces redundant testing cases so it is an effective method for automatic generation of testing cases.

Index Terms—Testing Cases; Basic Path; Control Flow; Triggering Path; Genetic Algorithm

I. INTRODUCTION

Software testing has become a necessary means for quality assurance in software production. Due to the appearance of object-oriented programming language, traditional testing methods cannot satisfy the need of object-oriented software testing [1]. The language features in object-oriented programming increase the complexity and enhance the testing difficulties. Automatic test of software is mainly used to find out modification errors of the software. Its operation procedures are similar to those before automatic testing, which offer automatic conditions for software testing. Even though a lot of research has been performed on object-oriented software testing at home and overseas. However, a set of mature and effective software testing theories and methods have not been formed currently. The newly introduced features of object-oriented technology bring new challenges for testing work [2]. On the other hand, automatic testing tools can effectively complete repetitive jobs. It cannot only reduce the testing cost, but it can also significantly increase the testing workload in limited time. In recent years, some references have offered discussions on object-oriented software testing. However, most of them are simple introductions for methods testing. Present research mainly contains two aspects [3-5]: One is object-oriented software testing based on programs; The other is object-oriented software testing based on protocol. Since these two object-oriented software testing methods are supplemented each other, some scholars suggest they should be combined to construct the testing methods of object-oriented software.

In terms of structural testing of object-oriented software, DONALD [6] proposed the increment testing technology based on class inheritance and she discussed how to repeat using superclass testing samples during the subclass testing, to improve the efficiency of object-oriented software testing. She also proposed a kind of testing method of class data stream. This method offers the concept of class control flow graph and proposes the idea to construct class control flow graph with source code. The data testing method adopts traditional data flow testing method to test the object-oriented programmes. D. c. Rankin [7] proposed an object state testing method based on programming. This method states that the object accuracy relies on whether object method can correctly modify the state of object. Then, it analyzed the source programming to construct testing model of the state. A testing-tree was constructed from the testing model of object state. Finally, a series of testing samples were constructed from the testing-tree. DeKung started from source program to construct the algorithm of object state testing model. On the basis of this algorithm, the supplementary testing tools can be developed to achieve semi-automation of object state testing method. In contrast, B. Haworth’s study [8] focused on traditional adequacy testing principle to construct class flow graph with data stream and control flow. Four adequacy principles of object testing based on class flow graph are defined and the inclusion relations between these adequacy principles are provided.
The overall target of object-oriented testing is to discover most errors with the minimized workload, which is consistent to the goal of traditional software testing. However, there have great differences between strategy and tactics on object-oriented testing. The concept of class in object-oriented programs is based on abstract data type with inheritance [9, 10]. Testing technology of the abstract data type is adaptive to class testing partially and it has disadvantages without taking into account the inheritance. In addition, since the class in object-oriented programs is short of clearly defined input/output behavior, the traditional testing process is only adaptive to class method testing, instead of the overall testing. The single test in class methods is not equal to class test. That is, to check correctness in class type singly cannot sufficiently guarantee the correctness on the whole. These problems exist with the appearance of object-oriented programming language and they are the focus of object-oriented testing research recently.

Figure 1. Note how the caption is centered in the column.

Class is the basic structural unit of object-oriented program. Only if correctness of class can be ensured, the correctness of system can be guaranteed. Object-oriented class testing is the key section of object-oriented testing. Thus, object-oriented testing technology is the key point in our research. Class testing can be divided into three parts according to the testing stages: testing based on service, testing based on state and testing based on response state. For each part of class testing, there are also two parts: structural testing, that is, path coverage and functional testing. Structural testing in this paper can be divided into two parts. One is the testing based on basic path and the other is based on genetic algorithm [11-13] to produce testing samples. On the basis of analysis on the effect of object-oriented programming language on testing, we discuss the overall testing process division of object-oriented programming and testing methods in different periods. Furthermore, the class service testing technology and automatic testing technology are studied in this paper. For the class service structure test, the designed control flow graph of block branch diagram is used at first, to determine the basic path of test. This method is an object-oriented automatic testing technology and its purpose is to produce testing samples which can evenly cover each dynamic performed path of the programs. Static control path in the program can be taken as referential basis and the path information will implement the equivalence partitioning. Fitness function of testing samples is designed by the number of elements in equivalence. By the adaptability of dynamic testing samples, redundancy generation of testing samples can be restricted. Meanwhile, it can promote to generate testing samples which is hard to generate paths. It can also effectively avoid testing redundancy and insufficiency caused by uneven distribution of testing samples space, which improves the automatic generation efficiency to a large extent.

II. TESTING CASE GENERATED BY BASIC PATH

A. Block Branch Diagram

For the testing model of class service, we adopt the Block Branch Diagram (abbreviated as BBD) proposed by Kung [14]. BBD of service f is a quintuple $BBD_f = [D_s, D_r, P, F_r, G]$. $D_s = [D_i \mid D_i]$ is the global data or class data introduced by $f$; $D_r = [D_i \mid D_i]$ is the global data or class data modified by $f$; $P = X, \theta_1, X, \theta_2, ..., X, \theta_n, X, \theta_i$ is the parameter table and function returned value of $f$. If $\theta_i$ is ↓ it denotes input, else if $\theta_i$ is ↑ it denotes output. When $X_i$ is default then there is no returned value. $F_r = [f_i \mid f_i$ is other services for call]. $G$ is a directed diagram named as block. It is base on the idea of control flow graph to modify the program flow of $f$, which represents the control structure of $f$.

$ \begin{align*}
D_s & \quad P \\
D_r & \quad F_r \\
G & 
\end{align*} $

Figure 2. BBD of change() of class ifelse

BBD sketch map of the method Compound conditional judgement in $f$ is decomposed and each judging frame has only single condition. To form execution paths later, we add digital mark to the initial node, ending node and each rectangle frame. BBD of class service is shown as figure 1.

The above class is written by C++. For the method change of class ifelse, its BBD is shown as figure 2. Generally to say, BBD should be constructed according to the testing demand, not source code, at software design stage. Because BBD constructed by source code can not
ensure the correctness. While BBD constructed at design stage can verify the program.

B. Testing Strategy

The class service are tested with statement coverage and branch coverage test through basic path [15, 16]. The basic path test requires to test $C$ different entrance-exit path. $C$ is the measurement of program and $C = e - n + 2$. $e$ is the number of IUT control graph and $n$ is the number of nodes. The formula derived from graph theory is called cyclometry complexity metrics, which works as single entrance.

Almost all the code coverage analyzer will generate this measurement. The structure test needs to generate control graph [17] first and then the basic testing path can be decided.

(1) Draw the control flow graph of class service as figure 3. The numeric marks correspond to those in figure 2;

(2) In the control flow graph of Ifelse::change, side $e = 9$, node $n = 7$ and measurement $C = 4$. This test needs 4 basic paths altogether. When the amount of paths is determined, we can find a group of testing path for all the nodes in traversing graph. According to figure 2, the basic path of structure test is shown as figure 3.

(3) Design testing cases to traverse these paths. Select appropriate data to satisfy the conditions of each basic path, which ensures every sentence on the path can be operated at least once. That is because the structure test is mainly used to coverage sentences. For each node on the control flow graph, the node which has the function of control has two entrances. Both of them must be traversed, that is, conditions “true” and “false” must be executed. The selection should ensure the two conditions be satisfied for the selection. Table 1 shows the basic path testing case of Ifelse::change.

![Figure 3. Control flow of change() of class Ifelse.](image)

III. TESTING CASES OF AUTOMATIC GENERATION BASED ON GENETIC ALGORITHM

A. Path Division

In the path-oriented testing cases automatic generation, all static control paths are needed to be calculated automatically first.

**Definition 1:** Set the directed adjacency matrix $G = \langle V, E \rangle$ is a directed graph. The finite set of nodes is $V = \{v_1, v_2, ..., v_n\}$; $A = (a_{ij})$ is an $m \times m$ matrix which satisfies

$$a_{ij} = \begin{cases} 1 & \text{if } v_i, v_j \in E \\ 0 & \text{if } v_i, v_j \notin E \end{cases}$$

(1)

Matrix $A$ is the directed adjacent matrix of graph $G$. This adjacent matrix can be acquired by covering-path of program AST after adjacent matrix is obtained, algorithm $PSCF_{path}$ is used to calculate all programming static control flow paths.

<table>
<thead>
<tr>
<th>Service name</th>
<th>IfElse::change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Path</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>Parameter Class data</td>
</tr>
<tr>
<td>Output</td>
<td>Parameter Class data</td>
</tr>
<tr>
<td>Path name</td>
<td>Input data Expected input data</td>
</tr>
<tr>
<td>Path 1</td>
<td>$a_1 = b_1, b_2, c_1, d_0 = 0$ Change $g_1 = a_1 = 0$</td>
</tr>
<tr>
<td>Path 2</td>
<td>$a_1 = b_1, c_1, d_2, e_1 = 1$ Change $g_1 = a_1 = 0$</td>
</tr>
<tr>
<td>Path 3</td>
<td>$a_1 = b_1, c_2, d_2, e_1 = 1$ Change $g_1 = a_1 = 1$</td>
</tr>
<tr>
<td>Path 4</td>
<td>$a_1 = 0$ Change $g_1 = a_1 = 1$</td>
</tr>
</tbody>
</table>

$PSCF_{path}$ automatically calculates all static control stream paths of graph $G$ with recursive pattern. When program instruction nodes are more and the path is complicated, the computing time and storage space of algorithm is increasing exponentially. But this pattern can perform automatic calculation of path which adapts to automatic testing process. It has practical value when program is not large. During practical calculation, through the patterns like dividing path DD [18] and compressing sentence nodes can divide structural programs. Each application algorithm $PSCF_{path}$ of sub structural module can be used for calculation actually.

Effective path of the programs is the control stream path acquired by calculating the adjacent matrix of program graph. Control stream path table $PSCF \_P$ constructed by all effective paths in the program is the basis of dynamic test. Normal path of programming dynamic performance should be stored in this table. But during actual testing process, the path in $PSCF \_P$ may not be complete. For example, the calculation in this algorithm can not cover all programming paths. This program may own a circle formed by cyclic path whose calculation is infinite. Algorithm $PSCF_{path}$ only calculates the cyclic path once to ensure that loop body can be operated at least one time. If all paths in table $PSCF \_P$ are covered, standard $C_2$ in [19] of coverage indicator will be satisfied by test. The other incomplete situation is, when there is mistaken program and the control transmission may be unusual. Then the program may execute the path which is not covered by $PSCF \_P$.

Based on $PSCF \_P$, testing triggering path appeared in practical operation in this program is defined.
Algorithm PSCF

Input: Initial node \( S_0 \) and triggering path \( P \)
Output: Control flow path table \( PSCF \_P \)
Variable: Adjacent matrix \( M_0 \)

Begin:

Add \( S_0 \) to \( P \) //add current node to triggering path
Record \( P \) to \( P' \) //record current triggering path
For \( j=0, j < n, j++ \) |
\( \text{if} ( M_0[S_k][j] \& \& \text{not in } P ) \)
\( \text{PS}_{PSCF}(j, P) \);
\( P++ \);
//continue to generate current triggering path
//trace back to previous branch node
Copy \( P \) to \( P' \);
Copy to \( P' \);
End

Definition 2: Assume the testing case is \( T \). Path \( P_i = \langle N_{i0}, N_{is}, ..., N_{in} \rangle \) is composed of a series of command nodes \( N_i \) which are executed sequentially under the function of testing case \( T \). Then path \( P_i \) is the triggering path of \( T \) and is recorded as 2-Tuple \( TP(T, P_i) \).

\( TP(T, P) \) depicts the path operation under specific testing cases. Triggering paths have three states:
- When triggering path \( TP \in PSCF \_P \) of testing cases is in the table of \( TP \in PSCF \_P \), it indicates that this path is a legal execution path.
- If triggering path \( TP \in PSCF \_P \) appears, it means that this path is abnormal execution of the program. It also means that this testing case triggers the errors of program operation.
- If there is a path which cannot be triggered by testing cases, \( PSCF \_P - \text{JP} \neq \phi \), it indicates the program may have infeasible path, which needs further analysis on the testing programs.

Theorem 1: We set the testing case set \( TP \) of all the trigger paths has formed a division in the zone input by testing programmes. The elements in \( TP \) have equivalence relation with triggering path \( P(x) \).

Proof: For \( \forall x, y, z \in TP(P) \) and \( x \neq y \neq z \)
(1) : Same testing case will trigger the same execution path
\( \therefore x = P(x) \) (Reflexive)
(2) If \( P(x) = P(y) \), then \( P(y) = P(x) \) (Symmetry)
(3) If \( P(x) = P(y) \) and \( P(y) = P(z) \), then \( P(x) = P(z) \) (Transitivity)
Above all, the division input by testing programs is the equivalence division of test-oriented triggering path. Different elements in the same kind of testing cases set will trigger the path owning the same operation.

B. Automatic Generation of Testing Cases

Definition 3: Set testing sample set of single triggering path \( P \) is \( T(P) \). The number of elements in this set is \( N \) and \( N \geq 0 \). Thus, the testing contribution of these elements to triggering path \( P \) is:
\[
C = \begin{cases} 
\frac{1}{N}, & N > 0 \\
0, & N = 0 
\end{cases}
\]

Elements in testing samples set \( T(P) \) have similar types to the path and the contribution of each element is the same for test. Elements in the set are expressed as the testing redundancy. If samples in testing with the same kind are more, it means this kind of test will be easier to appear in the test which is bad for generating testing samples, which can discover new path errors. Similarly, with increase of \( N \), the percentage to discovering the errors of programs for single test is lowering.

The main task using genetic algorithm to optimize triggering path, which is generated by testing samples, is to evenly generate testing sample set that can cover different execution paths. Since the distribution space and size of testing samples set on triggering path are different, it will result in unevenness of ransom generating testing cases and produce amounts of unnecessary redundant data for easily triggering path test case, with broad distribution space. However, for test case sets with narrow distribution space and difficult production, only minority testing data can be produced.

Genetic algorithm optimization is used to make up for the disadvantage for testing case space of uneven distribution. It generates reasonable testing cases as the partition of triggering path. Automatic generation of using genetic algorithm to acquire testing cases can be described as: Genetic algorithm is adopted to solve a group of optimized testing case. In each step of calculation process, the testing case automatic builder uses current colony to drive the execution of testing programmes. The triggering path of each testing case is tracked and recorded. Then the covering of maximize program execution path will provide calculation for adaptive target function, to generate the next generation of community. After multi-generation evolution it will go to the end when optimal populations are acquired or it exceeds certain circulation restrictions. The following algorithm is the description to get testing case automatic generation using genetic algorithm.

a) The first population is randomly generated in the input space of testing programs. During the design of initial weight, the path is final target and the triggering path set is initialized as \( PSCF \_P \). We assume the probability of each triggering path in this set is the same. So they have same initial weight \( 1/nP \). \( nP \) is the number of paths in triggering path set.

b) The current generated testing case \( T \) is taken as parent population to drive the testing programme and record the corresponding triggering path \( P \). The testing case generated randomly will be affected by the distribution of program. Therefore there is bigger
probability to generate similar or identical triggering path, reflecting the testing triggering path which is more easily to be executed in the programs. Then the testing cases which have the same triggering path are combined to generate the testing case set of current triggering path.

c) The corresponding fitness are re-calculated according to the contribution of single testing case to the triggering path. As mentioned earlier, the contribution describes the fitness of testing case to the test. The more contribution is, the more value it means to the test. In the process of calculation, all the testing cases in the set composed of the same kind of testing cases have the same contribution \( C_i = 1/n(P)N(T_i) \). \( C_i \) is directly used to express the fitness \( F_i \) of testing case.

d) Selection operator adopts the probabilistic methods of fitness proportion. It make choice according the proportion \( F_i / F \) between the fitness of individual in colony and the sum of individual fitness. So the individuals who have maximum fitness can be kept. Then matching crossover and mutation among these individuals with high fitness can be operated according to the proportion.

\[
F = \sum_{i=0}^{c} c_i, \quad i \in \text{triggering path and } T(P) > 0 \quad (3)
\]

e) Design of crossover operator adopts single-point crossover method. It makes the population to explore new gene space, which can create diversity new individuals. But it may destroy the patter of high fitness in the population and influence the performance in global search.

Variation is mainly reflected in the ability of local search in genetic space. When new path like \( P \notin \text{PSCF}_P \) emerges, the testing case caused such errors will make more local search to acquire better ability in error detecting.

f) The ending condition of algorithm is: in the testing cases set of all the triggering path, when the number of elements is not null and the fitness of testing case whose has the bigger fitness is lower than the threshold. The threshold should ensure that the amount of trigger paths which are most difficult to appeared reach a certain number.

C. The Error Detecting Ability of Testing Cases

Using genetic algorithm to automatically generate the testing cases and a group of complete set can be seen as a dynamic testing process. This process is a sample for the whole testing process of program. The statistical results will be acquired with the processes generated many times with genetic algorithm.

During the first time in execution process, we assume the generated triggering set is \( P_i = P_1 \cup P_2 \cup \ldots \cup P_m \). Corresponding testing cases set is \( T(P) = T(P_1) \cup T(P_2) \cup \ldots \cup T(P_m) \). If \( m \) tests are performed in this process, \( P = P_1 \cup P_2 \cup \ldots \cup P_m \) is used to describe all the triggering path sets. All the testing cases sets of triggering path form a partition in the input space of testing program. If \( T(P) \) is simplified as \( T_i \), then \( \sum T_i = T, \ T_i \cap T_j = \phi, \ i \neq j \).

Definition 4: Assume the triggering path set of in testing process is \( P \), the path coverage rate is \( \frac{P \cap \text{PSCF}_P}{P \cup \text{PSCF}_P} \).

Coverage rate reflects the degree of programmes dynamically executed with testing cases. \( P \cap \text{PSCF}_P \) refers to the unreachable paths that may exist in the programme while \( P \cup \text{PSCF}_P \) refers to the program error paths.

In the criteria of structural testing coverage, the path coverage has implied codes coverage. So the path coverage rate can be adopted for the statistics of testing effects. As referred in [20], over 90% code coverage rate can find out 70% errors of software, conservatively to say. However, this method can not process such testing methods for quantitative analysis. During the execution of each test the testing data is looked as a sample, so statistical method may be used in the program to analyze detected errors. The statistical detection methods is more complicated compared to coverage empirical methods. The related information of test subspace should be recorded and analyzed.

Definition 5: Let \( N(T_i) \) represent the number of elements in the subspace \( T_i \) of testing cases. \( n(T_i) \) refers to the number of actual testing case automatic generation. \( M(T_i) \) refers to the number of elements that may cased by program errors. Then the error density in the subspace \( T_i \) is \( \frac{M(T_i)}{N(T_i)} \).

Based on the error density, we can analyze the error detecting ability of automatic generation. The error detecting ability is described as

\[
\text{EDA}(G) = 1 - \prod_{r=0}^{n(i)} (1 - \frac{M(T_i)}{N(T_i)}) \quad (4)
\]

From experience to see, part of errors in the program can be detected and modified after each round testing case automatic generation. With the proceeding of this process the error density will also decrease. Since the testing case in each testing subspace increases \( n(i) \), the probability of unexposed error will decrease. When the error density supported by the testing sample data comes to a certain level, we can consider the demand of the test from a statistical point of view. So this kind of detecting method has higher credibility than path coverage rate method.

IV. EXPERIMENTS

A. Functional Test Based on Class

Structural test mainly refers to language coverage test. Covering a method and class cannot ensure that the software will not have errors. Because it only guarantees each code in the program can be operated but it cannot
guarantee completely correct function. Code analysis and code coverage can generally indicate an unsuitable testing package. We will have to perform functional test in order to guarantee there are no errors of class service in function. The basic method of functional test is to construct some reasonable input, to check whether the output can be matched to expected values.

When testing a function whose equivalence interval is (0,1) and (1, +∞), classical value 0.5 and 20 are adopted for equivalence partition tests. 0 and 1 are adopted for boundary value tests. After equivalence partition there may exists misunderstanding to acquire the middle value. So we should pay attention to some selection of classical value when selecting data. According to equivalence partitioning and boundary method, the functional test case for service change of class Ifelse is shown in table 2.

Since class Ifelse is only used to demonstrate functional testing data in both selection methods, it is meaningless for \( \text{Min}(\text{int}) \) and \( \text{Max}(\text{int}) \). For specific data, some are limited. So the choice for data case should be based on the boundary test, not just relying on simple data test. We can just test whether the service function can be performed correctly. The paths cannot be totally like structural test. Generally to say, the test we adopt mainly focus on the function. Because it has a direct impact on the results of program execution. But functional test can not ensure all the sentences be tested. Also, it cannot trigger the input error of program. Table 3 shows the testing results with genetic algorithm and randomly generated method when the number of colony is 100 and threshold is 0.0006.

### B. Error Detecting Ability of Testing Case

Triangle classification is used in this experiment and genetic algorithm is adopted to instruct the automatic generation of testing cases. Triangle classification program is described as: three input integers \( A, B, C \) are the sides of a triangle. The type of triangle is output according to the relation of sides. The testing case of triangle program has uneven distribution in input space. There are more internal branch paths and there is no loop block. It has better features for the detection of redundancy and fitness optimization. When using genetic algorithm, the problem will be encoded first. We set the range of triangle side \( A, B, C = [1,500] \) and we can directly use binary code. It is not complete encoding but it can trigger the input error of program. Table 3 shows the testing results with genetic algorithm and randomly generated method when the number of colony is 100 and threshold is 0.0006.
triangular, generally triangular, isosceles and equilateral triangle. The random mode run 547 times when \( PSCF_{\text{path}} \) is covered. The number of testing cases generated in narrow area of the input zone is smaller. Genetic algorithm can generate more testing cases on these paths, which ensure the sufficiency of test.

Figure 4 describes the size change of testing case set in different triggering paths with genetic algorithm and random method, during the evolution. (a) is random method and each round of testing case generation obey the independent identical distribution. The distribution of triggering path testing case set is similar to that of the input variable spatial data. Genetic algorithm has optimized the fitness well. When good testing case is generated its characters can be inherited. So bigger generation probability can be ensured in followed evolution of offspring, as is shown is (b).

![Figure 5. Relation between testing cases and testing coverage.](image)

Table 4 shows that this method can produce testing cases more automatically produce testing cases. The experiments number of testing cases generation can be regulated to divided into different partitions. Furthermore, on the basis of analysis on the effect of object-oriented programming language on testing, this paper discusses overall testing division of object-oriented program and the testing methods in different periods. In addition, this paper also studies the class service testing technology and the automatic testing technology. The structural testing mainly refers to basic path testing, that is, path coverage. Path coverage testing samples can be artificially produced and also be automatically generated. Artificial production is based on the artificial judgment to supply testing data and to traverse the paths. However, during the process of testing samples errors may occur. Therefore, automatic generation of testing samples can be constructed with genetic algorithm. According to the triggering path in dynamic operation, the generated testing cases are divided into different partitions. Furthermore, on the basis of fitness dynamics corresponding to the paths, the number of testing cases generation can be regulated to automatically produce testing cases. The experiments show that this method can produce testing cases more quickly and it has improved the testing efficiency.

### REFERENCES


