ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)
https://doi.org/10.17559/TV-20170319045945

LARGE SCALE SOFTWARE TEST DATA GENERATION BASED ON COLLECTIVE CONSTRAINT AND WEIGHTED COMBINATION METHOD

Dalin Zhang, Jianwei Sui, Yunzhan Gong

Original scientific paper

Software reliability test is to test software with the purpose of verifying whether the software achieves reliability requirements and evaluating software reliability level. Statistical-based software reliability testing generally includes three parts: building usage model, test data generation and testing. The construction of software usage model should reflect user's real use as far as possible. A huge number of test cases are required to satisfy the probability distribution of the actual usage situation; otherwise, the reliability test will lose its original meaning. In this paper, we first propose a new method of structuring software usage model based on modules and constraint-based heuristic method. Then we propose a method for the testing data generation in consideration of the combination and weight of the input data, which reduces a large number of possible combinations of input variables to a few representative ones and improves the practicability of the testing method. To verify the effectiveness of the method proposed in this paper, four groups of experiments are organized. The goodness of fit index (GFI) shows that the proposed method is closer to the actual software use; we also found that the method proposed in this paper has a better coverage by using Java Pathfinder to analyse the four sets of internal code coverage.

Keywords: *constraint; data generation; GFI; software reliability testing; usage model; weighted combination*

Generiranje ispitnih podataka za softver zasnovano na kolektivnom ograničenju i ponderiranoj metodi kombinacije

Izvorni znanstveni članak

Ispitivanje pouzdanosti softvera znači ispitivanje softvera kako bi se provjerilo da li udovoljava zahtjevima pouzdanosti i kako bi se procijenio njegov stupanj pouzdanosti. Statistički temeljeno ispitivanje pouzdanosti softvera općenito uključuje tri dijela: izgradnju modela, generiranje ispitnih podataka i ispitivanje. Stvaranje modela upotrebe softvera treba što je više moguće odražavati korisnikovu stvarnu primjenu. Potreban je ogroman broj ispitivanih slučajeva da bi se zadovoljila distribucija vjerojatnoće u slučaju stvarne upotrebe; inače će ispitivanje pouzdanosti izgubiti originalno značenje. U ovom radu najprije predlažemo novu metodu strukturiranja modela primjene softvera zasnovanu na modulima i heurističkoj metodi koja se temelji na ograničenjima. Zatim predlažemo metodu za generiranje podataka za ispitivanje uzimajući u obzir kombinaciju i težinu ulaznih podataka što smanjuje veliki broj mogućih kombinacija ulaznih varijabli na samo nekoliko reprezentativnih i povećava praktičnost primjene ispitne metode. U svrhu provjere učinkovitosti metode predložene u ovom radu, organizirane su četiri grupe eksperimenata. Ispravnost odgovarajućeg indeksa (GFI- goodness of fit index) pokazuje da je predložena metoda bliža upotrebi aktualnog softvera; također smo ustanovili da ima bolju pokrivenost kod uporabe Java Pathfinder-a za analizu četiri niza pokrivenosti internog koda.

Ključne riječi: *generiranje podataka; GFI; ispitivanje pouzdanosti softvera; model uporabe; ograničenje; ponderirana kombinacija*

1 Introduction

Reliability is a key indicator for the safe functioning of modern technological systems [1], such as air traffic control, railway transportation, and medical devices. Reliability is defined here as the probability of the failure-free operation of a software system for a specified period in a specified environment [2, 3]. Software reliability test (SRT) is to test software with the purpose of verifying whether the software achieves reliability requirements and evaluating software reliability level based on the operational profile which acquires failure data to estimate the reliability of a software product in quantifiable terms [4, 5]. A huge number of test cases with lengthy execution periods are currently required to satisfy the probability distribution of the actual usage situation. These test cases lead to a long execution cycle time of the SRT, a primary reason for the difficulties in applying SRT widely in engineering science today.

The most commonly used method of SRT is a statistical testing method based on the usage model, building software usage model and generating test cases based on the operational profile [6]. Markov model is the most widely used model, and the traditional test method is to generate a series of operation sequences through the Markov model [7]. The construction of software usage model should reflect user's real use as far as possible. A huge number of test cases are required to satisfy the probability distribution of the actual usage situation; otherwise, the reliability test will lose its original meaning. While using the Markov model to generate testing data, the most common method is one that generates testing data randomly [8]. However, this approach does not take the interaction among different operations into account, making this method generate redundant test data; meanwhile, different data may have different priorities according to the importance and using frequencies of the data. As a result, we need to assign weights to each type of input data, and higher weight means higher priority. Because of the above problems, in this paper, we first propose a new method of structuring Software usage model based on modules and heuristic method. Then we propose a method for the testing data generation in consideration of the combination and weight of the input data, which reduces a large number of possible combinations of input variables to a few representative ones and improves the practicability of the testing method. In order to verify the efficiency of our method, we perform four experiments. We compare the goodness of fit index (GFI) of our method with other methods in experiments. We also analyse the code coverage ratio of our method by using the tool Java Pathfinder. Results of these experiments show that our method could reduce the redundancy of test data and improve the testing efficiency while guaranteeing the coverage ratio of test data.

Our work focuses on improving the practicability by reflecting user's real use as far as possible in both the

stage of modeling and data generation stage. The main contributions of our work are: (1) we first propose a new method of structuring Software usage model based on modules and heuristic method; This method is more suitable for complex large-scale software systems; (2) Combination of three testing data generation technology, partitioning, combination and random. We firstly traverse all possible paths in the usage model and calculate the weight of each of them. We assume each type of input data is a discontinuous finite parameter (element) set. We assign a weight for every parameter. We propose a method for the testing data generation in consideration of the combination and weight of the input data, which reduces a large number of possible combinations of input; (3) In order to verify the effectiveness of the method proposed in this paper, four groups of experiments are organized. The GFI shows that the third method is closer to the actual software use; (4) we also found that the method proposed in this paper has better internal code coverage.

2 Usage models

2.1 Structuring the usage model

The Markov usage model (UM) can describe software usage scenario easily, its definition can be found in [7, 8, 9]. Researchers have proposed many kinds of methods of structuring usage model, which can be summarized to the following methods [9, 10, 11]: Musa's Method, based on expert experiences, historical data and complex software model. Musa's method is only a guiding thought of structuring usage model and lacks a specific implementation. The method of [9] is quite simple and cannot fit complex software. The method of [10, 11] cannot reflex calling relation and constraint relation among modules. In addition, with the growing of the quantity of modules, the complexity of this method is increasing sharply. To remedy these insufficiencies, this article proposes a method of structuring Software usage model based on modules and heuristic method. We need the following steps to structure a usage model for a complex software: (1) structuring the usage model under system (UMS); (2) structuring the usage models under module for every module (UMM); (3) finally, we get an UM through combining the UMS with the UMM via the module invoking state.

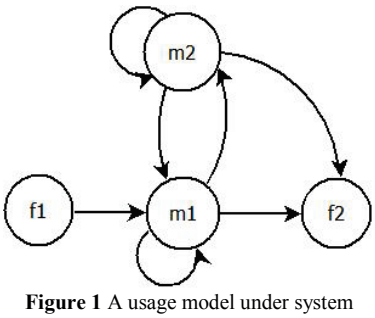

Figure 1 A usage model under system

In this paper, the UMS is a triple $\langle F, M, O\rangle$ and can be presented as a directed graph. As shown in figure 1, there is a set of points that can be expressed as a set of states, $F = \{f_1, f_2, \cdots f_n\}$,$M \subseteq F$is a set of module calling state, $M = \{m_1, m_2, \cdots m_n\}$; O is a set of edges and can be expressed as a set of operations, $O = \{fo_1, o_2, \cdots o_n\}$, $O \subseteq F \times F$. We add a new kind of states called "module calling" states. Take m_2 in figure 1 as an example; this state stands for the SUT invoking module 2 via its interface.

The Module usage model UMM is a triple $\langle F, M, O\rangle$ and can be presented as a directed graph. Unlike UMS, UMM can have multiple initial states which are determined by the module's interface (as shown in figure 3). As shown in Fig. 2, f_{13} is an initial state and indicates that the interfaces of this module m_1, and f_{25} and f_{26} are the initial states of module m_2. When UMS is under an invoking module state, it will search for the corresponding UMM by the name of the invoking module state, and then enter the UMM via the correct interface.

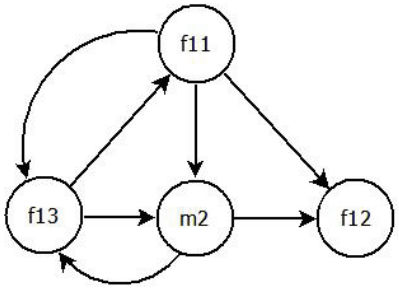

Figure 2 The usage model of model m_1

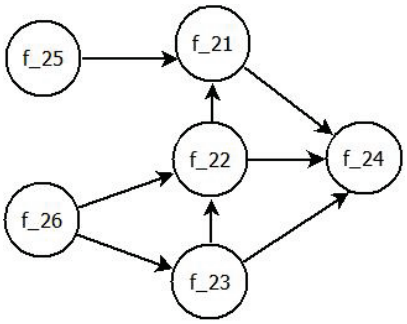

Figure 3 The usage model of model m_2

2.2 Transition probability

It is difficult to obtain the transition probability among states of an UM directly from historical data and experts; however, it is easy to obtain the constraint description (linear or non-linear) of each operation from an expert [11]. Then we calculate the operation transition probabilities according to constraints. Experts can offer constraints as follows:

Certain constraints: Such as $P(o_1) = 0.5$.

Linear constraints: Generally, the constraints in the linear function are the linear relation, include equality and inequality relation such as $0.3 \le P(o_1) \le 0.8$, $P(o_1) = 2P(o_2)$, $P(o_1) + P(o_3) = 0.6$, $P(o_3) < 3P(o_1)$.

Non-linear constraints: Such as $P(o_1) \le P(o_2)^2 P(o_1) = e^{P(o_2)}$.

The operation probabilities that belong to the same state should satisfy:

$$\sum_{i}^{n} x_i = 1, 0 \le x_i \le 1. \quad (1)$$

According to the principle of max entropy, the larger the entropy of a random variable is, the more objective what it reflects will be. We can use the principle of max entropy to calculate operation probabilities of UM: Under certain constraints, when the information entropy of transfer of states of UM is maximum, the UM is the closest to the actual usage of SUT, and this moment the operation transition probabilities are the ideal value that we want to calculate. After we obtain the constraints, we can calculate operation transition probabilities according to them. We convert the problem of calculating operation transition probabilities to an optimization problem as follows:

Maximum:

$$f(p) = \sum_{i=1}^{n} p_i log_2 p_i \tag{2}$$

Under Constraint:

$$\sum_{i}^{n} x_i = 1, 0 \le x_i \le 1. \tag{3}$$

We also use a genetic algorithm to solve this optimization problem and then we can get the synthetical operation transition probabilities as a significant property of UM.

The operation transition probabilities are calculated in this way, which is collected from a single expert. In order to make the result of calculation more objective, we need to synthesize the opinions of multiple experts. We adopt a method based on KL divergence to synthesize operation transition probabilities root in multiple experts [12]. KL divergence is relative entropy, it is an unsymmetrical measure of two random variables, the smaller the KL divergence is, the smaller will the discrepancy of the two variables be. Operation transition probabilities of UM are a discrete random variable. For two discrete random variables $X = \{x_1, x_2, \ldots x_n\}$ and $Y = \{y_1, y_2, \ldots y_n\}$, the KL divergence from X to Y and the KL divergence between X and Y are:

$$D_{KL}(X,Y) = \sum_{i=1}^{n} x_i \ln\left(\frac{x_i}{y_i}\right), D_{KL}(X,Y) \neq D_{KL}(Y,X) \tag{4}$$

$$D_{KL} = D_{KL}(X,Y) + D_{KL}(Y,X). \tag{5}$$

So, we can adopt D_{KL} to measure the discrepancy of X and Y.

According to the constraints offered by experts, we calculate out operation transition probabilities in-group of $P = \{P_1, P_2, \ldots P_n\}$, then we can search for a transfer probability C whose KL divergence to P is the smallest. This kind of transfer probability not only synthesizes opinions of multiple experts but also has the least discrepancy of all opinions of experts. In this way, we convert the problem of synthesizing multiple groups of operation transition probabilities to an optimized optimization problem as follows:

Let:

$$\mathrm{c} = \{c_1, c_2, \ldots c_n\}, P_j = \left\{p_1^j, p_2^j, \ldots p_m^j\right\}, j = 1, 2, \ldots, m \tag{6}$$

Minimum:

$$f(c) = \sum_{\mathrm{j}=1}^{n}\sum_{i=1}^{n}[c_i \ln\left(c_i / p_i^j\right) + p_i^j \ln\left(c_i / p_i^j\right)] \tag{7}$$

Under Constraint:

$$\sum_{i}^{n} c_i = 1, 0 \le c_i \le 1. \tag{8}$$

We also use a genetic algorithm to solve this optimization problem and then we can get the synthetical operation transition probabilities as a significant property of UM.

2.3 Types of input data and the interaction among operations

We define $K = \{k_1, k_2, \cdots, k_j\}$ as all the operation sequences (all paths in Markov model) in Markov model. If there are m operations in one operation sequence k_i, we can denote $k_i = \langle f_1, f_2, \cdots f_m \rangle$, and we can get finite set $F = \{f_1, f_2, \cdots f_n\}$, f_i denotes one operation in Markov model. In a finite set F, there are $a[i]$ types of input data in operation f_i. Then, we get finite set $V_i = \{1,2, \cdots a[i]\}, 1 \le i \le n$. We define n-gram testing sequence as follows:

$$Testseq = (v_1, v_2, \ldots, v_n), v_1 \in V_1, v_2 \in V_2, \ldots, v_n \in V_n. \tag{9}$$

$A = (a_{i,j})_{m \times n}$ is a $m \times n$ matrix, of which the j^{th} column represents the operation f_i of operation sequences. All the elements in the j^{th} column are from the collection V_j. Given a positive integer, if A can guarantee that any adjacent N (assuming $i, \cdots i+N-1$) columns can satisfy the condition that N-dimensional combination of elements in $V_i, \cdots V_{i+N-1}$ appears at least once, then we call A N-dimensional coverage array, and $NCA(m, N, F)$ for short. Each line of A represents the test data of the operation sequence. Apparently, m means the number of test data of the operation sequence.

Adjacent Matrix: We usually get the adjacent matrix according to the Markov model, and $p_{i,j}$ represents transition probability of point i to j. If there is no access from i to j, then we make $p_{i,j} = 0$.

The Weight of Operation Sequence: For an operation sequence $k_i, i \in [1, p]$, the p_j is the transition probability of the j^{th} operation sequence. We define the weight of operation sequence k_i as

$$Weight(k_i) = \prod_{i} p_i. \tag{10}$$

We assume each type of input data is a discontinuous finite parameter (element) set. We assign a weight $Weight(V_{i,k}) \in [0,1]$ for every parameter $V_{i,k}$ (the k^{th} value of collection V_i). The value of $Weight(V_{i,k})$ equals the probability of $V_{i,k}$ (the probability of the situation that the input data of $the\ i^{th}$ operation belongs to $the\ k^{th}$ type).

The weight of the parameter combination: while we make a N-dimensional adjacent combination for N parameters, the weight of the combination is

$$Weight_{combination} = \prod_{i=1}^{N} Weight(v_i). \quad (11)$$

In this paper, we use the directed graph to represent a usage model. Figure 4 shows all the states that start with an initial state and end with an ending state during the running process of vehicle software in hand-held terminal software developed by java. The edges in the graph represent operations of users and each edge is marked with a transition probability, which represents the probability of users, executes the corresponding operation.

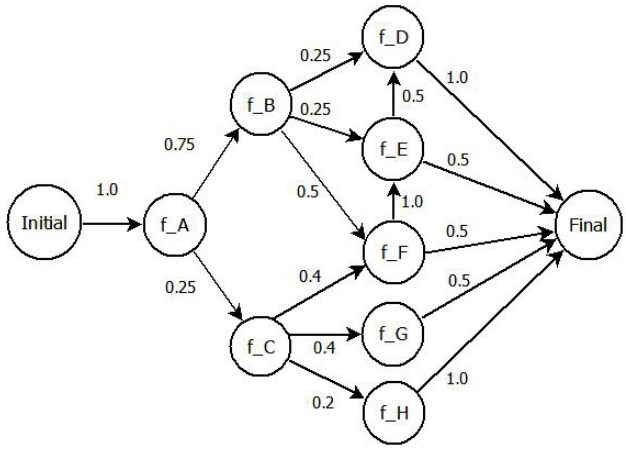

Figure 4 Markov usage model of vehicle software

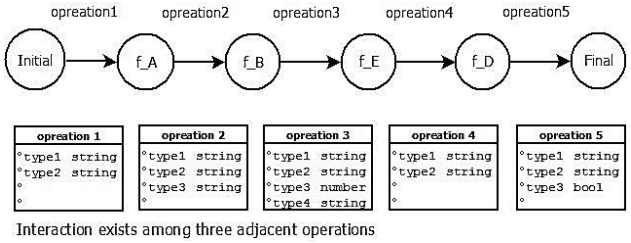

Figure 5 The operation sequence from Markov usage model

In an actual operation of software, as shown in figure 5, each step of the operation may have a variety of types of input data. For example, while classified according to data types, input data can be classified as integer, character and Boolean and so on. While it is classified by the effect of data, the input data also can be classified as valid data and invalid data. Meanwhile, there are often some interactions between different operations. Considering the diversity of input data and the interaction among operations, we need to generate test data for each operation sequence. Moreover, in the practical application of software, there are often strong interactions between adjacent operations, while there are often weak or even none interactions between non-adjacent operations. As a result, this focuses on the generating testing data method counting interactions between adjacent operations [13].

3 The algorithm of N-dimensional adjacent combination

3.1 Outline of method

This method can generate testing data and calculate the weight of testing data according to the Markov usage model. First, we get the adjacent matrix G of the Markov model. Then we use algorithm 1 to traverse all paths in the Markov model and calculate the probability of every path. As a result, we get all the possible operation sequences and weight them as follows:

$$K = \{k_1, k_2, \ldots, k_p\}, k_i, i \in [1, p]$$
$$Weight(K) = \{Weight(k_1), \ldots, Weight(k_p)\}. \quad (12)$$

For each k_i, we use algorithm 2 to make a N-dimensional adjacent combination of n operations in k_i. Then we could get the NCA of k_i. We get the proportion of each test data according to the formula:

$$proportion = weight_i / \sum weigth_i \quad (13)$$

3.2 Path generation algorithm

Algorithm 1 traverses every possible path in the usage model and calculates the weight of each path. Input parameters of function $getPaths()$ are the starting node number, the ending node number, the adjacent matrix G, the current path, and the current weight. For arbitrary node i in matrix G, there are three kinds of situations:

(1) There is no path between starting node and node i, then we ignore the node i keeping and look for next node.

(2) If node i is the ending node, this means that we have a path, and we need to record the path and the weight of path.

(3) If node i is not the ending node and there is a path between node i and starting node, this means that node i is one of the nodes in the path. Then we recursively call the function $getPaths()$ with node i being the value of parameter starting node.

Algorithm 1 Path Generation

Input:
the serial number of initial state: *start*;
the serial number of final state: *end*;
adjacent matrix *G* = {the adjacent matrix of Markov model};
initial *path*;
initial *weight* (default 0).
Output:
K = {all paths from initial state to final state};
Weight = {weight of all operation sequences}
getPaths(start,end,G,path,weight)
hasFlag[start]=true;
//set the *start* state visited
for (node *i* in adjacent matrix *G*) **do**
if*(G[start][i]==0 ||hasFlag[i])***then**
continue;
//if there is no path between *start* and *i* or *i* is already visited, go on and search next state.
end if
if (*i==end*)

```
        // if we have already found a path then
    record the result;
    continue;
    end if
getPaths(i,end,G, path+"→" +i,weight∗ G[start][i]);
        //keep searching
hasFlag[i]=false;
end for
```

3.3 N-dimension coverage algorithm

The output coverage array A of algorithm 2 is a matrix with m lines and n columns, and n is a constant which we have already known. At first, we need to calculate the scale of matrix A. And we get the parameter m by

$$m = \underset{1<=i<=n}{Max} \left(\prod_{k=i}^{k=i+N-1} a[k] \right). \tag{14}$$

Then we need to calculate the value of the first N columns by making the N-dimension combination of the first Ncolumns. Therefore, we get the value of the first N lines of matrix$A$. Based on this we need to extend our parameter to right. For example, after the first N lines are settled, we need to calculate the value of the N^{th} column. For parameterf_{N+1}, the number of times that each $N-1$dimension combination of $f_2, f_3, \cdots f_N$ occurs should be a_{N+1}. But the actual number of times that each $N-1$dimension combination of $f_2, f_3, \cdots f_N$ is a_1. If $a_1$is less than $a[N+1]$, we need to extend the number of times that each $N-1$ dimension combination of $f_2, f_3, \cdots f_N$ to $a[N+1]$. Then we traverse each $N-1$dimension combination of $f_2, f_3, \cdots f_N$ to get the value of the $(N+1)^{th}$ column. Similarly, we can get the value of all columns. It is obvious that each line in matrix A represents one of the testing data. At last, we calculate the weight and the proportion of each line.

Algorithm 2 N-dimension Coverage

Input:

operation sequence k_i ;

number of types of operation data in each operation $number = \{a[1],...,a[n]\}$;

$weight$ of every data type, dimension N .

Output:

N-dimension coverage array A .

for each element in array $number$ **do**

$m[i] = number[i] \times number[i+1]... \times number[i+N-1]$

end for

$m = \max(m[i])$;

//determine the number of NCA line

initial array $A[m][n]$;

// determine the scale of array A

calculate the element of the first N columns of A ;

// make a N-dimension combination

for $f_{N+1},...,f_n$ **do**

// make a right extend for the NCA

calculate the element of the nth column;

end for

calculate the total $weight$ of test data;

for every line of $A[m][n]$ **do**

calculate the proportion of each test data;

end for

3.4 Testing data evaluating

Different methods have different influence on a test, but the only evaluation criterion is whether the test data can reflect the actual operation accurately. Therefore, we import GFI to evaluate the method. We definite

$$GFI = \frac{\chi^2_{independent} - \chi^2_{proposed}}{\chi^2_{independent} - \chi^2_{ideal}} \tag{15}$$

$$\chi^2 = \sum_{i=1}^{n} \frac{(b_i - a_i)^2}{a_i}. \tag{16}$$

a_i represents the times that test case n occurs and b_i represents the times that test case i is assigned by some method. We assume that the $\chi^2_{independet}$ is the maximum value of $\chi^2$and there is the poorest correlation with reality. χ^2_{ideal}means the ideal condition (having the best correlation with reality).

According to the formula 15, it is not difficult to find that the value of χ^2_{ideal} should be 0. So, we can modify the formula GFI , and then we get a modified formula

$$GFI = \frac{\chi^2_{independent} - \chi^2_{proposed}}{\chi^2_{independent}}, \chi^2_{proposed} < \chi^2_{independent}. \tag{17}$$

It is evident that GFI is a constant between 0 and 1. A method is closer to reality while the $GFI$of the method is closer to 1.

4 Experiment

In order to verify the validity of the method proposed in this paper, we make experiments from the perspective of GFI and the coverage ratio of code.

4.1 Analysis of GFI

We design four groups of experiments to verify that the methods proposed in this paper have better GFI.

Method 1: We only use algorithm 1 to traverse the Markov model to get all operation sequences. We generate testing data for each operation sequence randomly without considering the diversity of input data and the interaction among operations.

Method 2: We use algorithm 1 and algorithm 2 to generate testing data for each operation sequence. This method considers the diversity of input data and the interaction among operations, but it ignores the weight of each operation sequence. It assigns testing data for each operation sequence averagely.

Method 3: On the basis of method 2, this method takes the weight of each operation sequence into consideration, and assigns testing data for each operation sequence according to the weight.

Method 4 (Corresponding to the Actual Situation in table 1): We import the actual usage statistics of a vehicle as the control group.

We take figure 2 as an example: the one of testing sequence is $\{initialstate, A, B, E, D, finalstate\}$, and the weight is $1 \times 0.75 \times 0.25 \times 0.5 \times 1 = 0.093750$. If we need to generate 100000 test data, and we know that this operation will need 9375 testing data.

We will evaluate methods in three groups by calculating GFI of three methods. For the operation in figure 2, there are 192 kinds of combination of testing data. According to our analysis, we can see that while considering the interaction among three adjacent operations, there are only 32 kinds of combination of testing data. We get the number 32 from the formula 14.

Table 1 The GFI of three methods

Sequencen umber	Method 1	Method2	Method3	The actual situation
1	97	292	159	141
2	48	292	159	169
3	146	292	243	280
4	48	292	243	221
5	48	292	159	131
6	0	292	159	177
7	0	292	243	291
8	97	292	243	300
9	146	292	121	100
10	0	292	121	121
11	48	292	178	191
12	146	292	178	177
13	48	292	375	400
14	48	292	375	420
15	0	292	562	625
16	0	292	562	611
17	0	292	375	310
18	146	292	375	301
19	48	292	562	675
20	97	292	562	500
21	48	292	281	330
22	0	292	281	210
23	48	292	421	374
24	48	292	421	272
25	48	292	121	122
26	48	292	121	130
27	48	292	178	183
28	0	292	178	184
29	0	292	281	283
30	0	292	281	267
31	97	292	421	430
32	48	292	421	449
total	1644	9344	9360	9375
GFI	0	0.602	0.965	1

This means that there are 160 invalid kinds of combination of testing data in the 192 kinds of combination of testing data. From table 1 we can see that method 1 assigns many testing data to the 160 invalid kinds of combination, which makes method 1 have the lowest efficiency. When compared to method 1, method 2 takes the interaction among three adjacent operations into consideration and assigns all testing data to the 32 valid kinds of combination. As a result, the method has a higher efficiency than method 1. However, there is also some disadvantages, the method assigns testing data averagely which makes number 9 and number 19 have the same number of testing data. However, this is quite different from the reality according to group 4. Moreover, method 3 improves this defect by considering the weight of each operation sequence. Method 3 assigns more testing data for the operation sequence with higher weight such as the number 19 and assigns less testing data for the operation sequence with lower weight such as the number 9. This makes method closer to reality.

χ_1^2, χ_2^2 and χ_3^2 represent the χ^2 of method 1, method 2 and method 3 separately. We can know that the maximum value is χ_1^2. So, we make $\chi_{independet}^2$ equal to χ_1^2 by calculating, and the GFI of each method is shown in table 1. $GFI_1 = 0$ shows that method1 is almost independent of reality due to ignoring the interaction among operation sequences; $GFI_2 = 0.6018$ shows that method 2 can partly reflect the reality; $GFI_3 = 0.965$ shows that method 3 is closest to the reality. Method 3 is the optimal method and is with the highest practical value.

4.2 Analysis of code coverage

The coverage ratio is a very important index in software testing. There are many kinds of coverage ratio, such as method coverage ratio, branch coverage ratio, and condition coverage ratio. In this part, we use the tool Java Pathfinder [14] to analyse the code coverage ratio from the perspective of method coverage ratio and branch coverage ratio.

The JavaPathfinder is used to find defects in Java programs, so you also need to give it the properties to check for as input. The JavaPathfinder gets back to you with a report that says if the properties hold and/or which verification artefacts have been created by the Java Pathfinder for further analysis. So, we need to generate testing data as input data of the program under test by ourselves. We run the program on the Java Pathfinder to find defects in the program. In order to evaluate our method, we generate testing data by our method (method 3). And we get a report about the method coverage ratio and the branch coverage ratio from the Java Pathfinder.

Table 2 The coverage ratio of code

Function number	Covered function	Function coverage ratio
23	21	91.3 %
Branch number	Covered branches	Branch coverage ratio
58	51	87.9 %

As is shown in table 2, method coverage ratio could reach 91.3 %, while the branch coverage ratio could reach 87.9 %. We guarantee both the method coverage ratio and the branch coverage ratio with less testing data by using method 3.

5 Relation work

5.1 The construction of software usage model

Random testing selects test data uniformly at random from the input domain of the program. When the random selection is based on some operational profile, it is sometimes called statistical or operational testing and can be used to make reliability estimates [4, 15].

Statistical testing can help software testing and be used to assess software reliability. The construction of software usage model should reflect user's real use as far as possible. A huge number of test cases are required to satisfy the probability distribution of the actual usage situation; otherwise, the reliability test will lose its original meaning. Researchers have proposed many kinds of methods of structuring usage model, which can be summarized to the following methods: Musa's Method [9], based on expert experiences [11], historical data and complex software model [16]. Musa's method is only a guiding thought of structuring usage model lacking a specific implementation. The method of [9] is quite simple and cannot be applied to complex software. The method of [11] cannot reflex calling relation and constraint relation among modules. In addition, with the growing of the quantity of modules, the complexity of this method is increasing sharply.

Different from the above methods, we propose a new method of structuring software usage model based on modules and heuristic method. It is difficult to obtain the transition probability among states of an UM directly from historical data and experts; however, it is easy to obtain the constraint description (linear or non-linear) of each operation from an expert. The opinions from multiple experts should be synthesized to make the result of calculation more objective. We synthesize the transition probabilities among multiple experts based on KL divergence [12].

5.2 Test case generation

Statistical testing generates test data by sampling from a probability distribution defined over the software's input domain. The distribution is chosen carefully so that it satisfies an adequacy criterion based on the testing objective, typically expressed in terms of functional or structural properties of the software. A huge number of test cases are currently required to satisfy the probability distribution of the operational profile and the actual usage situation. These test cases lead to a long execution cycle time of the SRT, the primary reason for the difficulties in applying SRT widely. How to choose a few test values from a large data space is important for SRT.

The most common method is the one that generates testing data randomly [15]. However, this approach does not take the interaction among different operations into account, making this method generate redundant test data. The basic idea is to split the data space into equivalence classes and choose one representative from each equivalence class, with the hope that the elements of this class are equivalent in terms of their ability to detect failures [4]. Pairwise and N-way coverage criteria [17] are popular forms of data coverage criteria. Combination strategies are test case selection methods that identify test cases by combining values of the different test object input parameters based on some combinatorial strategy [13]. In combinatorial testing, the issue is to reduce a large number of possible combinations of input variables to a few representative ones.

Boundary analysis and domain analysis are widely accepted as fault detection heuristics and can be used as coverage criteria for test generation [18, 19]. For ordered data types, the partitioning of a range of values into equivalence classes is usually complemented by picking extra tests from the boundaries of the intervals [20, 21].

Test data are generated by sampling from a probability distribution chosen so that each element of the software structure is exercised with a high probability. However, deriving a suitable distribution is difficult for all but the simplest of programs. The [22] demonstrates that automated search is a practical method of finding near-optimal probability distributions for real-world programs, and that test sets generated from these distributions continue to show superior efficiency in detecting faults in the software.

Considering the above issues, our work focuses on improving the method by considering the interaction among different operations and the weight of operation data. We propose a method for the testing data generation in consideration of partitioning (weighted parameters), combination and random. We first traverse all possible paths in the usage model and calculate the weight of each of them. We assume each type of input data can be denoted by a discontinuous finite parameter set. We assign a weight for every parameter. Our method reduces a large number of possible combinations of input. We also found that the method proposed in this paper has a better coverage by using the Java Pathfinder to analyse the four sets of internal code coverage.

5.3 Model-based testing

At present, there are many kinds of model-based testing tools for reliability estimate. The JUMBL [23] is an academic model-based statistical testing tool. Test inputs are generated by traversing the usage model while respecting transition probabilities in JUMBL: the test cases with the greatest probability are generated first. While using the Markov model to generate testing data, the most common method is one that generates testing data randomly [24-26]. However, those approaches do not consider interaction among different operations and the weight of testing data.

The shortest searching paths are found and the redundant test sequences are reduced based on the natural law of ants foraging in W. Zheng's works [27]. Different from it, we consider not only the test sequence problem but also the test data problem. At the same time, we give the modelling method for large-scale software. As described in this paper, the Algorithm 1 traverses every possible path in the usage model and calculates the weight of each path.

In L. Fernandez-sanz's works [28], the specifications considered the logical starting point to generate a set of test cases which covers most of the functional testing needed to validate a software product. The research [28] is a typical method for automatically generating a

complete set of functional test cases from UML activity diagrams. At the same time, there is also a prioritization according to software risk information in L. Fernandez-sanz's works. Different from Fernandez-sanz's works, our research focuses on software reliability testing and verifying whether the software achieves reliability requirements and evaluates software reliability level. We try to generate test cases to satisfy the probability distribution of the actual usage situation.

6 Conclusion and future work

In this paper, from the perspective of engineering application, we focus on improving the practicability by reflecting user's real use as far as possible both in the stage of modelling and data generation stage. We first propose a new method of structuring Software UM and calculate its transition probability based on modules and heuristic method. In the stage of data generation, we propose the method that takes the interaction among operations and the weight of operation sequence into account. However, there are also some disadvantages in our method. Our method performs well when it is applied to the software that interaction only exists in adjacent operations. But, it will lead to degradation in performance when our method is applied to the software where interaction does not only exist in adjacent operations. This limits the applicability of our method, and we will improve it in future.

Acknowledgment

This work is sponsored by the National Natural Science Foundation of China under Grant No.61502029. We are grateful to the anonymous reviewers for their comments on earlier drafts of this paper.

7 References

[1] Zio, E. Reliability engineering: old problems and new challenges. // Reliability Engineering& System Safety. 94, 2(2009), pp. 125-141. https://doi.org/10.1016/j.ress.2008.06.002

[2] Reussner, R. H.; Schmidt, H. W.; Poernomo, I. H. Reliability prediction for component-based software architectures. // J. Systems Softw. 66, 3(2003), pp. 241-252. https://doi.org/10.1016/S0164-1212(02)00080-8

[3] Immonen, A.; Niemel, E. Survey of reliability and availability prediction methods from the viewpoint of software architecture. // Software & Systems Modeling. 7, 1(2008), pp. 49-65. https://doi.org/10.1007/s10270-006-0040-x

[4] Ai, J.; Lu, M.; Ruan, L. Usage profile construction technique for generation of software reliability test data. // Jisuanji Gongcheng Computer Engineering. 32, 22(2006), pp. 7-9.

[5] Nelson, E. Estimating software reliability from test data. // Microelectronics Reliability. 17, 1(1978), pp.67-73.

[6] Trammell, C. Quantifying the reliability of software: statistical testing based on a usage model. // Software Engineering Standards Symposium, Proceedings of software engineering standards symposium / Montreal, Que., 1995, pp. 208-218. https:// doi.org/10.1109/sess.1995.525966

[7] Whittaker, J. A.; Thomason, M. G. A Markov chain model for statistical software testing.// IEEE Transactions on Software engineering. 20, 10(1994), pp. 812-824. https://doi.org/10.1109/32.328991

[8] Walton, G. H.; Poore, J. H.; Trammell, C. J. Statistical testing of software based on a usage model. // Software: Practice and Experience. 25, 1(1995), pp. 97-108. https://doi.org/10.1002/spe.4380250106

[9] Musa, J. D. Operational profiles in software-reliability engineering. // IEEE software.10, 2(1993), pp. 14-32. https://doi.org/10.1109/52.199724

[10] Takagi, T.; Furukawa, Z. Construction technique of large operational profiles for statistical software testing. // Computer and Information Science. 2013, pp. 187-199. https://doi.org/10.1007/978-3-319-00804-2_14

[11] Poore, J. H.; Walton, G. H.; Whittaker, J. A. A constraint-based approach to the representation of software usage models. // Information and Software Technology. 42, 12(2000), pp. 825-833. https://doi.org/10.1016/S0950-5849(00)00101-4

[12] He, Y.; Zhou, D.; Wang, Q. Research on aggregated approach to group decision making with fuzzy judgment matrix. // Chinese Journal of Management Science. 2, (2008).

[13] Wang, Z. Y.; Nie, C. H.; Xu, B. W. Generating combinatorial test suite for interaction relationship. // Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting / New York, 2007, pp. 55-61. https://doi.org/10.1145/1295074.1295085

[14] Pasareanu, C. S.; Rungta, N. Symbolic PathFinder: symbolic execution of Java byte code. // Proceedings of the IEEE/ACM international conference on Automated software engineering / Antwerp, Belgium, 2010, pp. 179-180. https://doi.org/10.1145/1858996.1859035

[15] Soto, J. Statistical testing of random number generators. // Proceedings of the 22nd National Information Systems Security Conference / Gaithersburg, 1999.

[16] Smidts, C.; Mutha, C.; Rodríguez, M.; Gerber, M. J. Software testing with an operational profile: OP definition. // ACM Computing Surveys. 46, 3((2014), pp. 39-49. https://doi.org/10.1145/2518106

[17] Grindal, M.; Offutt, J.; Andler, S. F. Combination testing strategies: a survey. // Software Testing, Verification and Reliability. 15, 3(2005), pp. 167-199. https://doi.org/10.1002/stvr.319

[18] Hamlet, D.; Taylor, R. Partition testing does not inspire confidence (program testing). // IEEE Transactions on Software Engineering. 16, 12(1990), pp. 1402-1411. https://doi.org/10.1109/32.62448

[19] Gutjahr, W. J. Partition testing vs. random testing: The influence of uncertainty. // IEEE Transactions on Software Engineering. 25, 5(1999), pp. 661-674. https://doi.org/10.1109/32.815325

[20] Zhang D.; Jin D.; Gong Y.; Wang C.; Chen S. Research of alarm correlations based on static defect detection. // Technical Gazette. 22, 2 (2015), pp. 311-318. https://doi.org/10.17559/TV-20150317102804

[21] Wang Y. W.; Xing Y.; Gong Y. Z.; Zhang X. Z. Optimized branch and bound for path-wise test data generation. // International Journal of Computers Communications & Control. 9, 4(2014), pp. 497-509. https://doi.org/10.15837/ijccc.2014.4.1169

[22] Poulding, S.; Clark, J. A. Efficient software verification: statistical testing using automated search. // IEEE Transactions on Software Engineering. 36, 6(2010), pp. 763-777. https://doi.org/10.1109/TSE.2010.24

[23] Prowell, S. J. JUMBL: A tool for model-based statistical testing. // System Sciences, Proceedings of the 36th Annual Hawaii International Conference on / Big Island, USA, 2003. https://doi.org/10.1109/HICSS.2003.1174916

[24] Lin, L.; He, J.; Song, F. Usage modeling through sequence enumeration for automated statistical testing of a GUI application. // Software Engineering and Service Science, Proceedings of 2014 IEEE 5th International Conference on / Beijing, China, 2014, pp. 82-85. https://doi.org/10.1109/ICSESS.6933518

[25] Utting, M.; Pretschner, A.; Legeard, B. A taxonomy of model-based testing approaches. // Software Testing, Verification and Reliability. 22, 5(2012), pp. 297-312. https://doi.org/10.1002/stvr.456

[26] Homm, D.; Eckert, J.; German, R. Combining time and concurrency in model-based statistical testing of embedded real-time systems. // International Conference on Software Engineering and Formal Methods, 2015, pp. 22-31. https://doi.org/10.1007/978-3-662-49224-6_3

[27] Zheng, W.; Hu, N. Automated test sequence optimization based on the maze algorithm and ant colony algorithm. // International Journal of Computers Communications & Control. 10, 4(2015), pp. 593-606. https://doi.org/10.15837/ijccc.2015.4.1732

[28] Fernandez-Sanz, L.; Misra, S. Practical application of UML activity diagrams for the generation of test cases. // Proceedings of the Romanian Academy, Series A. 13, 3(2012), pp. 251-260.

Authors' addresses

Dalin Zhang, PhD, *Corresponding author*
National Research Center of Railway Safety Assessment,
Beijing Jiaotong University, Beijing, China
E-mail: dalin@bjtu.edu.cn

Jianwei Sui, doctoral student
State Key Laboratory of Networking and Switching,
University of Posts and Telecommunications, Beijing, China
E-mail: jwsui@bupt.edu.cn

Yunzhan Gong, Professor
State Key Laboratory of Networking and Switching,
University of Posts and Telecommunications, Beijing, China
E-mail: gongyz@bupt.edu.cn