International Journal of Control and Automation
Vol. 9, No. 12 (2016), pp.35-50
http//dx.doi.org/10.14257/ijca.2016.9.12.04

Automatic Test Case Generation and Optimization Strategies based on Components with Sequential Constraints

Shiwei Gao[1], Jianghua Lv[1], Fan Yang[1], Jiaqi Xie[1] and Shilong Ma[1]

[1]*State Key Laboratory of Software Development Environment, Beihang University, No. 37, Xueyuan Road, Haidian District, 100191 Beijing, China*
ge89@163.com, jhlv@nlsde.buaa.edu.cn, gaiyangfan@163.com, libcanyoung@gmail.com, slma@nlsde.buaa.edu.cn

Abstract

Combinatorial testing has been proved to be an efficient strategy in software testing. The research on automatic test case generation for system level based on components with constraints is a challenging but significant problem by using combinatorial testing. This paper focuses on the test case generation for system level based on components with sequential constraints. There are three main contributions: 1) the new test case generation strategy for system level is proposed based on IPOG (In-Parameter-Order-General) algorithm; 2) a filling strategy for don't care positions is given to enhance the higher strength coverage for system level, which can improve the quality of the generated test suite; 3) a randomized post-optimization algorithm is applied to reduce the size of the test suite for system level. Experimental results show that the proposed filling strategy can improve the quality of the covering array, the post-optimization algorithm has a good optimization effect on reducing the size of the array, and trades off better between the size of the covering array and computational time than simulated annealing (SA).

Keywords*: Combinatorial testing, Software testing, Automatic test case Generation, IPOG, Randomized post-optimization*

1. Introduction

Many complex systems today are built using highly integrated components, which creates new challenge for integration testing. Take spacecraft automatic test system for example, as typical safety-critical systems, their trustworthiness is very important. Once the system does not work, it will cause property, life and other significant losses. Generally speaking, spacecraft automatic test system is complex system, which is composed of many kinds of functional units constructed by the high integration of multiple components. And the reliability of system is supported by effective and adequate component testing. Naturally, it is critical for obtaining accurate and full testing results. Therefore, the research on component testing has very important significance.

System faults are often caused by unexpected interactions among components [1]. It is nearly always impossible to exhaustively test all of these interactions among components because of resource constraints. Thus, we need a strategy to construct a subset of these interactions. Combinatorial testing [2, 3, 4, 5], which has been widely studied and proved to be an effective test strategy, can help to resolve the problem. The key problem for combinatorial testing is the automatic test case generation by constructing covering arrays. Covering arrays have been used for software interaction testing by D.Cohen *et al*. in the Automatic Efficient Test Generator (AETG) [6], and the corresponding

ISSN: 2005-4297 IJCA

automatic test case generation tool has been developed. A covering array, *CA(n;t,k,v)* is a $N \times k$ array such that every $N \times t$ sub-array contains all ordered subsets from *v* symbols of size *t* at least once. The covering array number is the minimum *N* required to satisfy the parameters *t, k, v*, and *t* is the strength [6]. Suppose we have a power switch control system with on-off switches, and there are about 34 switches. If we want to test this system by exhaustive testing, there are $2^{34} = 1.7 \times 10^{10}$ possible inputs = 1.7×10^{10} tests, while only 33 tests are needed if we use 3-way combinatorial testing. Kuhn *et al*. [7] studied the faults in several software projects, and found that all the known faults are caused by interactions among 6 or fewer parameters, which can be seen easily from Figure 1, which shows the cumulative percentage of faults at $t = 1$ to 6 for various applications. Therefore, both of the strength for component and system level is t ($2 \leq t \leq 6$), and the strength of system level is less than or equal to that of the component level.

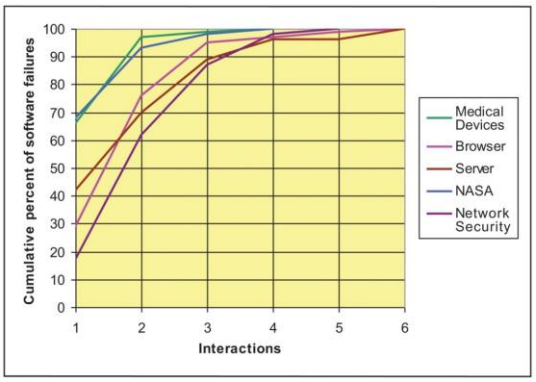

Figure 1. Cumulative Fault Distribution

An automatic test case generation strategy, which could generate test cases for system level among components by using the computational method of simulated annealing algorithm, was proposed by Cohen et.al [6, 8]. This strategy would resolve the automatic test case generation problem for component level with the same strength and system level with different strength. However, the different strength for component level and the sequential constraints among components are not considered, which indeed exist in the real life systems. Therefore, existing methods are not enough for a system level testing, especially for complex system. Firstly, some components are so critical for a system that they should be tested with higher coverage strength. Secondly, one or more sequential constraints among components are prevalent, system faults are often caused by unexpected sequential constraints among components in the actual testing, including serial, parallel and loop constraints.

To resolve these problems in the above, first of all, the sequential constraints among components are represented by a directed graph, and a path on the directed graph shows a sequential constraint. The *VCAP* is given to denote the covering array with sequential constraints, which converts the problem of automatic test case generation into constructing *VCAP*. Secondly, the strategy for constructing *VCAP* is given based on IPOG [9] algorithm. The proposed strategy respectively satisfies *t*-way interactions testing for both system and component levels. At last, randomized post-optimization algorithm [10, 11] is used to reduce the size of the covering array for system level.

The rest of this paper is organized as follows: Section 2 describes the related work briefly. Section3 gives the related definitions and problem analysis. Section 4 presents the strategy for constructing test cases based on components with sequential constraints. Section 5 applies the randomized post-optimization algorithm to reduce the

size of the covering array for system level. Section 6 shows the experiments results of the coverage rate and size of the covering array. Section 7 concludes the paper with a summary and a description of future work.

2. Related Work

Combinatorial testing has been topics of research for many years, and many important research results are gained.

Combinatorial testing can detect failures triggered by interactions of parameters (components) in the system under test (SUT) with a covering array test suite generated by some sampling mechanisms [12]. Test case generation is the most active area of combinatorial testing research. There are many automatic test case generation strategies. Up to now, there are four main groups of methods which have been put forward: greedy algorithm, heuristic search algorithm, mathematic method and random method. Greedy algorithms have been the most widely used method for test suite generation. There are two classes of greedy algorithms. The first class is the one-test-at-a-time based on the automatic test case generator AETG [13]. Bryce *et al*. gave the density-based greedy algorithm for constructing a 2-way and higher strength covering array in [14, 15]. The first class greedy algorithms can be combined with heuristic search techniques to reduce the size of covering array at the cost of time. In [6, 8], variable covering array (*VCA*) is constructed by using simulated annealing (SA) algorithm. It is a pity that the sequential constraints are not considered. The second class of greedy algorithm is the IPO algorithm, which begins with generation all t-sets for the first *t* parameters and then incrementally expands the solution, both horizontally and vertically using heuristics until the array is complete. The IPO algorithm is proposed to generate 2-way covering array by Lei el al. in [16]. Then the IPOG algorithm is given to generate *t*-way covering array in [9]. IPOG-D [17], IPOG-F and IPOG-F2 algorithms [18, 19, 20], which are the variation of IPOG, are put forward. Generally speaking, the first class greedy algorithm is able to construct a relatively small size of covering arrays at the cost of time, while the second class greedy algorithm is able to construct a test suite with trivial covering arrays but less time overhead. In this paper, test case generation strategy is mainly based on the IPOG algorithm and some adjustment is made to fit our needs.

Constraint handling problem are very important in combinatorial testing. For this problem, the constraints usually are converted to logical expression and solved by calling SAT solver. For example, zChaff is used to solve pairwise test case generation in presence of constraints in [21], and Choco solver [22] is used to handle constraints for higher strength test case generation. However, these strategies focus on constraints among parameters. Therefore, these strategies could not be applied efficiently to solve the constraints among components.

Combination coverage is an important metric to measure the quality of the covering array, especially for higher strength coverage. A special metric for (t+1)-way coverage is useful, which can help to design a covering array with high quality. A new metric for combinatorial test suites is proposed in [23]. And some research about combinatorial coverage is conducted in [24, 25, 26, 27]. Besides, the corresponding combinatorial coverage measurement tool has been developed [28]. Therefore, it is necessary to consider the strength coverage rate in designing the test case generation strategy. In order to enhance the higher strength coverage, a filling strategy is introduced in this paper.

The most ideal situation is that the optimal size of covering array is generated with the smallest time overhead, which is difficult to achieve. The reason is that the test case generation problem is NP-hard. Therefore, we hope that a relatively smaller size of covering array can be constructed by using the proposed strategy in reasonable time.

Randomized post-optimization algorithm, which has been proved to be an easy and effective means to improve a wide variety of covering arrays, was proposed to reduce the size of the generated covering array [10, 11]. Randomized post-optimization algorithm does not depend on a particular construction technique and provides a relatively fast method for detecting and exploiting duplication of coverage in order to improve the arrays. Therefore, the randomized post-optimization algorithm is used to reduce size of the generated covering array for system level in this paper.

3. Problem Definition and Analysis

3.1. Problem Definitions

Definition1 Component: Let Component=<ID, P>,P={p_1, p_2, …, p_k}, p_i=<name, V_i>, V_i={v_1,v_2,…, v_n}, where, ID refers to the component's identifier, *k* is the number of parameter for a component, name is the parameter name, V is the value set of a parameter.

Definition2 Sequential Constraints (SC): Given two components of $Component_1$ and $Component_2$, let SC= $Component_1 \preccurlyeq Component_2$, where $\preccurlyeq$ denotes that $component_2$ has a higher priority compared to $Component_1$.

Definition3 System Under Test (SUT): SUT= < {Component}, {SC}>.

Definition4 Path: Path={$Component_1 \preccurlyeq Component_2 \preccurlyeq \ldots \preccurlyeq Component_n$ }. A path means that a kind of sequential constraint. Where, *n* is the number of components on a path.

A variable strength covering array, denoted as a $VCA(N;t,(v_1,v_2,\cdots v_k),C)$, is an $N \times k$ mixed level covering array, of strength *t* containing C, a multi-set of disjoint mixed level covering arrays each of strength $\geq t$. This definition of *VCA* is limited to describe a system consisting of multiple components. *VCA* is unable to describe the sequential constraints among components. In order to make up the above shortcoming, the definition of *VCA* is modified as follows.

Definition 5 A variable strength covering array with path (*VCAP*): $VCAP(N;t,(v_1,v_2,\cdots v_k),path)$ is a $N \times k$ mixed level covering array (*MCA*), of strength t containing path, which shows the sequential constraints among components. Where, the strength may differ from each other for every component. Take $VCAP(N;t,MCA(3,3^2 2^4) \preccurlyeq MCA(4,4^2 3^1 2^3) \preccurlyeq MCA(3,2^3 4^2))$ for example, the SUT consists of three components, and the sequential constraint is described as $MCA(3,3^2 2^4) \preccurlyeq MCA(4,4^2 3^1 2^3) \preccurlyeq MCA(3,2^3 4^2)$

A variable strength covering array with path *VCAP* means a test suite based on a path for a SUT. Normally a directed graph has only several paths. But the paths contained on the graph would be countless if there is a circle on the graph. A *threshold* will be given to limit the number of paths according to the shortest path length of the directed graph without repeated nodes. How to automatically generate the system level test cases for all the paths on the graph is the problem to resolve in this paper.

3.2. Problem Analysis

A simple example will be given to elaborate the problem of system level test cases generation based on components with sequential constraints using the related definitions in the above.

Given a system under test, there are four function modules. Each module is made up of one or two components, each module has five parameters, and each parameter has three values. The sequential constrains among the four components are listed as follows:

$(A_1, \{<p_1,\{v_1\}>, <p_2,\{v_1,v_2\}>, <p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (B, \{<p_1,\{v_1\}>, <p_2, \{v_2, v_3\}>,<p_3,V_3>, <p_4, V_4>, <p_5, V_5>\})$;

$(B_1, \{<p_1,\{v_1\}>, <p_2, \{v_2, v_3\}>,<p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\})$;

$(A_1, \{<p_1,\{v_1\}>, <p_2,\{v_1,v_2\}>, <p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (C, \{<p_1, \{v_1, v_3\}>, <p_2, \{v_2,v_3\}>,<p_3,V_3>, <p_4, V_4>, <p_5, V_5>\})$;

$(C_1, \{<p_1,\{v_1,v_3\}>, <p_2,\{v_2,v_3\}>,<p_3,V_3>, <p_4,V_4>, <p_5,V_5>\}) \lessdot (C_2,\{<p1,\{v_1\}>, <p2,\{v_3\}>, <p3,\{v_2\}, <p_4, V_4>, <p_5, V_5>\})$;

$(C_2, <p1,\{v_1\}>, <p2,\{v_3\}>, <p3,\{v_2\}, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\})$;

$(D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\})$;

The sequential constraints is shown as Figure 2, where, a module is denoted as a rectangle, a component is denoted as an oval, and the arrow is denoted as a sequential constraint between adjacent components. It should be noted the fact that P^A is a set of parameters which belong to A.

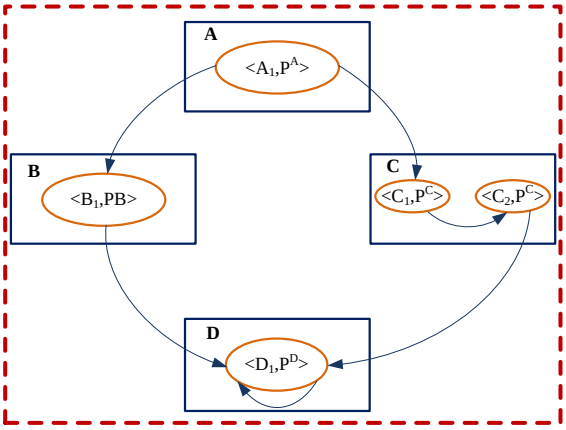

Figure 2. Sequential Constraints

Then the sequential constraints among components can be converted into a directed graph. As shown in Figure 2, the sequential constraints among components can be described by the paths on the directed graph. There is a circle on the graph, which will result in countless paths. According to the actual needs, it is possible to avoid the situations of countless paths by setting the path length. We can refer it as *threshold*. With different *threshold*, different set of paths will be obtained. But the *threshold* is no less than the shortest path length on the graph without repeated nodes.

Two test paths can be obtained from Figure 2 if the threshold is set as 3, which are listed as follows:

$(A1, \{<p_1,\{v_1\}>, <p_2,\{v_1,v_2\}>, <p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (B_1, \{<p_1,\{v_1\}>, <p_2, \{v_2, v_3\}>,<p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\})$;

$(A_1, \{<p_1,\{v_1\}>\}, <p_2,\{v_1,v_2\}>, <p_3,V_3>, <p_4, V_4>, <p_5, V_5>\}) \lessdot (C_1, \{<p_1,\{v_1,v_3\}>, <p_2,\{v_2,v_3\}>,<p_3,V_3>, <p_4,V_4>, <p_5,V_5>\}) \lessdot (C_2,\{<p1,\{v_1\}>, <p2,\{v_3\}>, <p3,\{v_2\}, <p_4, V_4>, <p_5, V_5>\}) \lessdot (D_1, \{<p_1,\{v_2\}>, <p_2, \{v_2, v_3\}>, <p_3, \{v_1, v_3\}>, <p_4, V_4>, <p_5, V_5>\})$;

The two paths can be described by using the definition of $VCAP(N;t,(v_1,v_2,\cdots v_k),path)$:

$VCAP(N;t,1^4 2^6 3^{10}, MCA(t_1,1^1 2^1 3^3) \preccurlyeq MCA(t_2,1^1 2^1 3^3) \preccurlyeq MCA(t_3,1^1 2^2 3^2) \preccurlyeq MCA(t_4,1^1 2^2 3^2))$;

$VCAP(N;t,1^5 2^5 3^{10}, MCA(t_1,1^1 2^1 3^3) \preccurlyeq MCA(t_3,2^2 3^3) \preccurlyeq MCA(t_4,1^3 3^2) \preccurlyeq MCA(t_4,1^1 2^2 3^2))$;

Therefore, the problem of test case generation based on components with sequential constraints is converted into constructing a variable covering array with path $VCAP(N;t,(v_1,v_2,\cdots v_k),Path)$. A test suite, which satisfies the given constraints, will be constructed for each path. As shown in Figure 2, it is not difficult to find that there are serial, parallel and loop constraints among components, which will be able to be processed by our given strategy. The three kinds of constraint can be seen as from the following Figure 3:

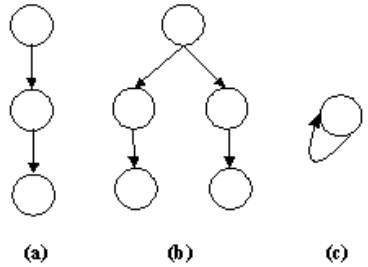

Figure 3. Three Kinds of Constraint: Serial, Parallel and Loop Constraint

4. Our Method

For a given SUT, the sequential constraints among components will be firstly obtained. Then a directed graph is able to be created according to the sequential constraints among components. It is naturally to set the *threshold* in case of countless paths if there is a circle on the graph. Now all of the paths on the graph will be generated, which means all the sequential constraints among components for the system are built. A kind of sequential constraint is denoted by a path on the graph. A strategy for constructing variable strength covering array with path (*VCAP*) will be given in details in the following discussion.

Variable strength covering array with path (*VCAP*) can be viewed as a collection of mixed covering arrays inside of a larger covering array. In order to construct the variable strength array with path (*VCAP*), we use IPOG algorithm to find a lower strength mixed covering array for the whole system and alter it to obtain the higher strength coverage required for the designated component subsets. First of all, we briefly give the framework of IPOG algorithm.

For a system with *n* parameters, where $n \geq t$, the IPOG algorithm first builds a covering array for the first *t* parameters, and then extends the array for the rest $n - t$ parameters, one parameter at a time, until the covering array is complete. The extension for each of the $n - t$ parameters is achieved by two steps: horizontal growth and vertical growth. Horizontal growth adds a value of the new parameter to each existing row so that this extended row covers the most value combinations of in π. π represents the set of uncovered *t*-way combinations of values involving this new parameter and $t - 1$ parameters among the already included parameters. Vertical growth adds new rows, if necessary, to cover remained value combinations in π which have not been covered during horizontal growth.

The flow chart of IPOG algorithm is shown in Figure 4.

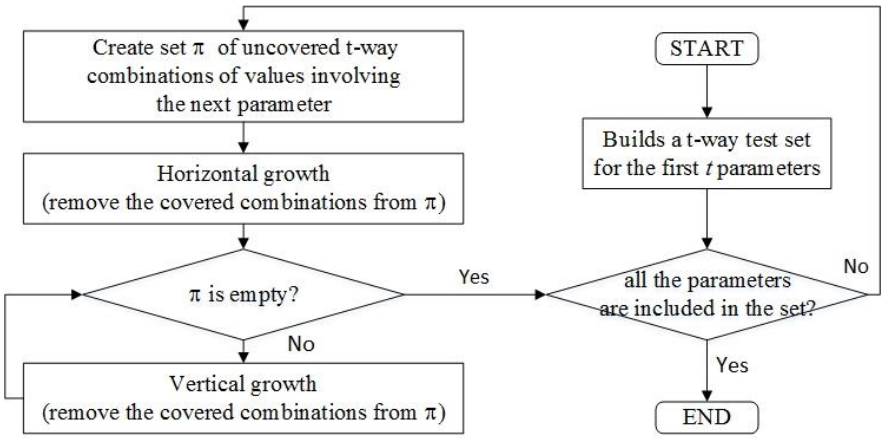

Figure 4. Flow Chart of IPOG Algorithm

Based on the IPOG algorithm, the main steps of the test cases generation strategy for system level with sequential constraints is given as follows:

Step1. In the initialization phase of our strategy, obtain the strength coverage of the system level, the number of component, the parameters, the values for each parameter, the higher strength coverage for each component and the sequential constraints among components. Let the strength coverage of the system be *t*. And *m* is the number of components and $t_k(1 \le k \le m)$ is the higher strength coverage for the corresponding component.

Step2. Convert the sequential constraints into a directed graph.

Step3. Set the *threshold* for the path length.

Step4. Get the set of paths by traversing the directed graph on the basis of the *threshold*, and let *n* be the number of paths.

Step5. Select a path, set $n=n-1$, and calculate the number of components on the path. Let *p* is the number of components, and *i* is the counter with initial value 1.

Step6. Construct a lower strength *t* covering array for the whole system by using IPOG.

Step7. Construct a higher strength mixed covering array for the *i*th component with its strength coverage and priority on the path by using IPOG algorithm.

Step8. Add the mixed covering array for the *i*th component to the covering array for the whole system constructed in Step6, and set $i=i+1$.

Step9. If $i>p$, go to Step10; otherwise, go to Step7.

Step10. Fill the *don't care* positions with values for the corresponding parameter in the covering array for system level, and output the covering array for the selected path.

Step11. If $n=0$, stop the procedure; otherwise, go to Step5.

Where, *don't care* positions are generally referred to the entries in test set matrix of which values have not been determined, and they are usually temporarily noted as a certain symbol like '-' to defer this determination until more information is available.

4.1. Constructing Covering Array for System Level

First of all, the lower strength covering array for the whole system with a sequential constraint (path) would be constructed by using IPOG algorithm. The basic idea is that the whole system could be considered as one component with strength *t* (the lower strength for the whole system), and all the parameters of the whole system with different level of values also could be regarded as in the component. Therefore, the IPOG algorithm would be directly used to construct the mixed covering array for the system level.

4.2. Constructing Covering Array for Component Level

The higher strength mixed covering arrays for component level are built successively according to the sequential constraint (path) by using IPOG algorithm after generating the lower strength covering array for the system level. When a higher strength mixed covering array for a component is constructed, it must be added to the lower mixed covering array for the system level, which simultaneously satisfies both of the strength for the system and component level. Repeat this operation until the mixed covering arrays for all of the components on the basis of the priority on the path are built.

It is apparent that part of the higher strength interactions for the component level has been included in the lower strength mixed covering array for system level. Therefore, the construction of mixed covering array for component level is an extension of the covering array for the system level. It will guarantee that the uncovered interactions for the strength of the component must be added to the mixed covering array for the system level.

The pseudocode of constructing mixed covering arrays for component level is stated as follows:

```
1. suppose that there are k components on a path, construct the mixed covering arrays according
to the componet's order on the path, let ts be the mixed covering array for system level
2. for(int i=1;i≤k;i++)
3. {
//construct mixed covering array for the ith component on the path, t_i is the strength of the ith
component, n_i is the number of parameters for the ith component, ts_i is mixed covering array of the ith
component
4. initialize ts_i to be an empty set
5. sort the parameters of the ith component in an increasing order of their domain sizes
6. find all of the uncovered interactions of the first t_i parameters in ts
7. vertical growth for the first t_i parameter, and add the uncovered interactions to ts_i
8. for(int j=t_i+1;j≤n_i;j++)
9. {
10. horizontal growth for the jth parameter of the ith component
11. vertical growth for the jth parameter of the ith component
12. }
13. recover the original order of the ith component
14. add ts_i to ts
15. }
```

4.3. Filling Strategy

Because the size of the mixed covering array may differ from each component, there will be many *don't care* positions after all of the components with sequential constraint being added to the mixed array for the system level. Selecting the values of these *don't care* positions for the corresponding parameter will affect neither of the strength coverage for system and components. However, that will have an effect on the higher strength coverage for the system level. Therefore, it is better to raise the higher strength coverage for the system level in a real test environment.

The traditional method is selecting the value with the least number of the values for the corresponding parameter for filling *don't care* positions. In this paper, an improved filling strategy is given as follows:

If *don't care* position is in the odd-numbered line, the corresponding parameter will be assigned within domain of the parameter randomly. Otherwise, if *don't care* position is in the even number line, the corresponding parameter will be assigned with the least number of the values for the parameter.

There are two advantages for our filling strategy. On the one hand, the number of value combinations for the higher strength will be increased, which means the higher

strength coverage for system level enhanced. On the other hand, it makes the distribution of the value for all of the parameters more reasonable, which does benefit for the sufficiency of testing. The proposed filling strategy improves the quality of the test suite for the system level.

The pseudocode of our filling strategy is given as follows:

```
let ts be the mixed covering array for the system level
Calculate the line number of ts, and for the N
for(int i=1;i≤N;i++)
{
   if(there exists don't care positions in the ith line){
      if(i is an odd number)
         all of the don't care positions are assigned with value of the corresponding parameter randomly
for the ith line
      else
         all of the don't care positions in the ith line are assigned with the least number of the values for
the corresponding parameter
 }
output ts
```

5. Optimization Strategy

A covering array for the system level based on the sequential constraints among components is constructed in the above. Randomized post-optimization algorithm is used to reduce the size of the mixed covering array for the system level.

5.1. Randomized Post-Optimization Algorithm

It is necessary to review several concepts in post-optimization proposed in [10, 11]. A position in an array A= (a_{ij}) is a pair(r, c), in which r is a row index and c is a column index. An entry in A is a triple(r, c, s) where (r, c) is a position, and a_{rc}=s. Every entry of a CA (N; t, k, v) participates in $\binom{k-1}{t-1}$ t-way interactions. When all of the $\binom{k-1}{t-1}$ t-way interactions involving a specific entry are covered more than once, the entry can be changed arbitrarily, or indeed omitted in the determination of coverage, and the array remains a covering array. Such a position is a flexible position or possible *don't care* position. Accordingly, a flexible set of positions is a set for which all entries could be omitted in the determination of coverage, and the array remains a covering array. A nominated row in an array is a row that has the most flexible positions.

Post-optimization strategy is a heuristic method with randomness. The main idea of post-optimization strategy is that repeatedly adjusts the array by eliminating some duplication of coverage for t-way interactions in an attempt to reduce its number of rows in a given time. The covering array retains full coverage during the adjusting process. Each time for the adjusting process is seen as an iteration process. At each iteration process, all rows in the array except the last are randomly reordered, and the flexible set is found. At the same time, the nominated row is searched by traversing the array. If all of elements in the nominated row are included in the flexible set, the nominated row can be deleted. Repeat this iteration process until the number of iterations permitted without improvement is reached in the given time. One iteration process of post-optimization algorithm is stated as follows:

Step1: Find the flexible set F and mark the corresponding flexible position. Firstly, all the positions in the array are marked as unnecessary. Secondly, each row in the array is checked for every *t*-way combination. If a combination for the *t*-way interaction is found for the first time, the *t* positions for this combination will be modified as necessary. Repeat this process until all *t*-way combinations are marked. Then the set of all the unnecessary is the flexible set.

Step2: Find the nominated row. If all of elements in the nominated row are included in the flexible set, the nominated row can be deleted. Otherwise, continue finding the nominated row until one row cannot be deleted directly. Then put the nominated row at the end of the array.

Step3: Create flexible positions in the nominated row. Let F be a flexible set; let F'' be the positions of F in the nominated row, and let $F' = F - F''$. For each entry f=(r, c, s) in F', consider the column c in which it appears. If the nominated row has entry s in column c, and this position is not in F'', replace the entry f with symbol s. If the position in column c of the nominated row is in F'', choose an entry at random to replace entry f.

Step4: Reorder except the last row in the array.

5.2. Refining the Mixed Covering Array for System Level

As mentioned in the above, the strength for system level is less than or equal to that of the component level. At the same time, the strength of components may differ from each other. Thus, post-optimization algorithm cannot be used to optimize the covering array for system level directly. The strength for system and component level must be considered together. A simple strategy is given as follows:

Step1: all the positions in the mixed array for system level are marked as unnecessary.

Step2: find all the combinations of strength *t* for system level and when a combination is covered in the array for the first time we mark its t positions as necessary.

Step3: mark all positions of every component as unnecessary.

Step4: find all the combinations of strength *t'* for every component and when a combination is covered in the corresponding array for the first time we mark its *t'* positions as necessary.

Step5: all of the positions marked as unnecessary belong to the flexible set. Then the post-optimization algorithm can be used to reduce the size of array.

In a word, the mark of combinations for each component is increased when we use post-optimization algorithm to optimize the array for system level.

6. Experiment Analysis

To evaluate the performance of the proposed strategies, the experiments are conducted. Our experiments have three goals. Firstly, we want to study the coverage rate change on the higher strength coverage for the system level by using our filling strategy. Secondly, we want to compare the size of the array for system level before and after using post-optimization algorithm. Thirdly, we give a comparison on the size of constructing *VCAP* and the computational time between our strategy and SA. All of the experimental results reported here are for a core(TM)2 Duo Intel(R) processor clocked at 2.93GHz, and 2GB memory.

6.1. Coverage Rate

A covering array that provides 100% simple combinatorial coverage for *t*-way combinations will also provide some degree of coverage for (*t*+1)-way combinations, (*t*+2)-way combinations, *etc*. For some applications, it may be useful to enhance the higher strength coverage. We use the method given in [28] to calculate the higher strength coverage rate.

The proposed filling strategy will raise the higher strength coverage for the system level. In order to prove this, given four simple systems with one sequential constraint, the coverage rates for system level with strength from 2 to 6 by using the existing and our proposed strategies are reported respectively. The four given systems are listed as follows:

$VCAP_1(N;2,2^{10}3^5 5^5, MCA(3,2^5 3^1 5^1) \preccurlyeq MCA(5,2^3 3^3 5^3) \preccurlyeq MCA(3,2^2 3^1 5^1))$,

$VCAP_2(N;3,2^{10}3^5 5^5, MCA(4,2^5 3^1 5^1) \preccurlyeq MCA(5,2^3 3^3 5^3) \preccurlyeq MCA(3,2^2 3^1 5^1))$,

$VCAP_3(N;3,3^{10}4^8 7^5, MCA(4,3^4 4^4 7^1) \preccurlyeq MCA(5,3^3 4^3 7^2) \preccurlyeq MCA(4,3^3 4^1 7^2))$,

$VCAP_4(N;4,4^8 7^5 3^4 5^7, MCA(6,4^3 7^1 3^2 5^2) \preccurlyeq MCA(5,4^2 7^3 3^1 5^3) \preccurlyeq MCA(4,4^3 7^1 3^1 5^2))$.

The first system $VCAP_1$ has three components with given sequential constraint $MCA(3,2^5 3^1 5^1) \preccurlyeq MCA(5,2^3 3^3 5^3) \preccurlyeq MCA(3,2^2 3^1 5^1))$ and strength 2. The first component with strength 3 has seven parameters: five parameters have two values, one parameter has three values, and one parameter has five values. The second component with strength 5 has nine parameters: three parameters have two values, three parameters have three values, and three parameters have five values. The third component with strength 3 has four parameters: two parameters have two values, one parameter has three values, and one parameter has five values. The detailed information for the other components is similar.

Table1 shows the sizes of the test suites generated by using the existing strategy and our filling strategy respectively when filling *don't care* positions, and the time taken to generate these test suites. Table2 shows the changes of the coverage for the system level with strength from 2 to 6 by using the existing and our filling strategies respectively.

Table 1. Results of Four Systems under Test with Sequential Constraints

VCAP	**Strategy**	**Size of test suite**	**Time**
$VCAP_1$	Existing strategy	1444	0.281s
	Our Strategy	1444	0.281s
$VCAP_2$	Existing strategy	1481	0.547s
	Our Strategy	1481	0.547s
$VCAP_3$	Existing strategy	4132	2.016s
	Our Strategy	4132	2.016s
$VCAP_4$	Existing strategy	14855	92.532s
	Our Strategy	14855	92.984s

Table 2. Changes of the Higher Strength Coverage Rate for the System Level with Strength from 2 To 6

	$VCAP_1$		$VCAP_2$		$VCAP_3$		$VCAP_4$	
Strength	**Existing strategy**	**Our Strategy**	**Existing strategy**	**Our Strategy**	**Existing strategy**	**Our Strategy**	**Existing strategy**	**Our Strategy**
2-way	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
3-way	**98.03%**	**100.00%**	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
4-way	**89.17%**	**99.87%**	**96.00%**	**99.74%**	**93.55%**	**99.62%**	100.00%	100.00%
5-way	**72.24%**	**95.76%**	**80.25%**	**95.09%**	**65.02%**	**87.38%**	**95.72%**	**97.31%**
6-way	**48.44%**	**76.66%**	**53.30%**	**76.13%**	**29.12%**	**48.51%**	**61.67%**	**66.49%**

It is apparently that the filling strategy will not affect the size of test suite. And the effect on the time taken to generate the test suites is very weak, which can be seen from Table1. Therefore, we can ignore the time influence of the two strategies. Seen from Table2, for $VCAP_1$ with strength 2, the coverage rates for higher strength from 3 to 6 are obviously improved by using our strategy. At the same time, the changes of higher strength coverage rates for the other given systems are similar. We can say that the proposed filling strategy is able to improve the higher strength coverage for the

system level, which means the quality of the test suite for system level can be improved without increasing time cost and size of test suite.

6.2. Size of Test Cases

The post-optimization algorithm will reduce the size of covering array for the system level generated based on the components with sequential constraints in reasonable time. In order to prove this, one simple system $VCAP_5$ with one sequential constraint is given. The system $VCAP_5$ with sequential constraint is given as follows:

$$VCAP_5(N;2,3^4 5^7 7^5 9^2, MCA(3,3^2 5^2 7^2 9^1) \preccurlyeq MCA(4,3^2 5^3 7^2 9^1) \preccurlyeq MCA(2,5^2 7^1)).$$

The system has three components with one given sequential constraint $MCA(3,3^2 5^2 7^2 9^1) \preccurlyeq MCA(4,3^2 5^3 7^2 9^1) \preccurlyeq MCA(2,5^2 7^1)$ and strength 2. The first component with strength 3 has seven parameters: two parameters have three values, two parameters have five values, two parameters have seven values, and one parameter has nine values. The second component with strength 4 has eight parameters: two parameters have three values, three parameters have five values, two parameters have seven values, and one parameter has nine values. The third component with strength 2 has three parameters: two parameters have five values, and one parameter has seven values.

Firstly of all, the size of covering array for system level with strength 2 is 2777 and time taken to generate is 0.422s by using the proposed strategy in this paper based on IPOG algorithm. Then the post-optimization algorithm runs ten times to reduce the size of the generated covering array for system level with the given time 10s, 20s, 30s, 40s, 50s, 60s respectively. Table3 gives the minimum, maximum and average sizes obtained after 10 runs of post- optimization algorithm for the given system with the given time 10s, 20s, 30s, 40s, 50s, 60s. Meanwhile, the corresponding number of iterations is also shown with the minimum, maximum and average number of the iterations. Finally, the reduction rate is calculated.

It can be seen from Table 3 that the size of covering array obviously decreases with the growth of execution time of post-optimization algorithm and the increasing of iteration times. The size of the array will be reduced in further if the given time is increased according to the actual test requirement. The reduction rate has something about the given system. If the size of the array has been close to optimal, the optimization effect will not be obvious. However, post-optimization algorithm provides a relatively fast method for detecting and exploiting duplication of coverage in order to improve the size of the array. Therefore, we can say that the post-optimization algorithm is able to reduce the size of covering array for the system level generated based on the components with sequential constraints in reasonable time, and has a good optimization effect.

Table 3. Result of Size of Covering Array For System Level and the Corresponding Numbers of Iteration After 10 Runs of Post-Optimization With 10s, 20s, 30s, 40s, 50s and 60s Respectively

optimization Time	Size of covering array			Number of iteration			Reduction Rate
	Min	Max	Avg	Min	Max	Avg	
10s	2697	2703	2700.2	130	131	130.8	**2.77%**
20s	2640	2654	2645.1	263	264	263.8	**4.75%**
30s	2603	2615	2606.6	390	402	399	**6.14%**
40s	2565	2582	2578.5	535	541	539.1	**7.15%**
50s	2552	2569	2560.2	677	680	678.3	**7.81%**
60s	2528	2552	2540.5	815	820	817.9	**8.52%**

6.3. Comparison with Simulated Annealing

Cohen [6] reported using simulated annealing (SA) to construct variable strength array without considering the sequential constraints among components and different strength for each component, While the proposed strategy based on IPOG algorithm is able to generate variable strength array with path (*VCAP*) and satisfy *t*-way interactions testing for both system and component levels. Obviously, our strategy can construct variable strength array. A comparison between our strategy and simulated annealing on the size of test cases for the given systems will be given in this section. According to the definition of *VCAP*, the selected systems [6] can be described as follows:

$VCAP_6(N;2,3^{15},\phi)$,

$VCAP_7(N;2,3^{15},CA(3,3^4) \preccurlyeq CA(3,3^5) \preccurlyeq CA(3,3^6))$,

$VCAP_8(N;2,3^{15},CA(3,3^6) \preccurlyeq CA(3,3^9))$,

$VCAP_9(N;2,3^{15},CA(3,3^7) \preccurlyeq CA(3,3^8))$,

$VCAP_{10}(N;2,3^{15},CA(3,3^9) \preccurlyeq CA(3,3^6))$,

$VCAP_{11}(N;2,4^3 5^3 6^2,CA(3,4^3) \preccurlyeq MCA(3,5^6 6^2))$,

$VCAP_{12}(N;2,4^3 5^3 6^2,CA(3,5^3) \preccurlyeq MCA(3,4^3 6^2))$,

$VCAP_{13}(N;2,4^3 5^3 6^2,MCA(3,5^1 6^2) \preccurlyeq MCA(3,4^3 5^2))$,

$VCAP_{14}(N;2,4^3 5^3 6^2,MCA(3,4^3 5^3 6^2))$,

$VCAP_{15}(N;2,3^{20}10^2,CA(3,3^{20}) \preccurlyeq CA(3,10^2))$.

Table 4. Comparison between Proposed Strategy and SA on Size for *VCAP* after 10 Runs

VCAP	Strategy	Min	Max	Avg.	Optimization time
$VCAP_6$	Our strategy	16	17	16.2	1s
	SA[6]	16	17	16.1	NA
$VCAP_7$	Our strategy	35	37	35.8	30s
	SA[6]	33	35	34.8	NA
$VCAP_8$	Our strategy	33	38	36.4	30s
	SA[6]	33	35	34.9	NA
$VCAP_9$	Our strategy	44	48	45.8	1s
	SA[6]	41	42	41.4	NA
$VCAP_{10}$	Our strategy	51	53	52.5	5s
	SA[6]	50	51	50.8	NA
$VCAP_{11}$	Our strategy	64	64	64	1s
	SA[6]	64	64	64	NA
$VCAP_{12}$	Our strategy	125	125	125	1s
	SA[6]	125	125	125	NA
$VCAP_{13}$	Our strategy	180	180	180	1s
	SA[6]	180	180	180	NA
$VCAP_{14}$	Our strategy	214	216	215.4	30s
	SA[6]	214	216	215	NA
$VCAP_{15}$	Our strategy	100	100	100	10s
	SA[6]	100	100	100	NA

Table 4 presents the minimal (Min), the maximal (Max) and the average (Avg.) size of *VCAP* generated by our strategy and simulated annealing [6] ,respectively, after ten independent runs. Where, NA means that the specific time overhead is not given in [6].

It seems like that there is no obvious advantage on size the *VCAP* constructed by our strategy, especially for $VCAP_7$, $VCAP_8$, $VCAP_9$, $VCAP_{10}$, and $VCAP_{14}$,while there are almost the same size for $VCAP_6$, $VCAP_{11}$, $VCAP_{12}$, $VCAP_{13}$ and $VCAP_{15}$. The gap is not large between our strategy and simulated annealing, because the worst case is only 4.4(45.8-41.4=4.4) for $VCAP_9$. At the same time, the computational time also should be considered. As described in [6], the first few *VCAP*'s in Table4 complete in seconds, while the larger problems, such as the last *VCAP* in Table 4, complete within a few hours. However, most of results reported by using our strategy are completed in seconds. Therefore, we can say our strategy trades off better between the size and computational time than SA.

7. Conclusion

A new combinatorial object, $VCAP(N;t,(v_1,v_2,\cdots v_k),Path)$, which can be used to define the test suite based on the components interaction with sequential constraint. A feasible strategy is proposed to construct it. Therefore, the test suites for system level based on the components with sequential constraints can be constructed. Then a filling strategy is proposed to improve the higher strength coverage for system level, which can obviously enhance the quality of the covering array without additional cost. Finally, the post-optimization algorithm is used to reduce the size of the array for system level. The experimental results show that the proposed strategy can generate test cases based on components with sequential constraints efficiently in a reasonable size and computational time.

Acknowledgements

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61300007 and 61305054, the Fundamental Research Funds for the Ministry of Science and Technology of China under Grant No. YWF-14-JSJXY-007, and Free exploration projects for State Key Laboratory of Software Development Environment under Grant Nos. SKLSDE-2015ZX-09 and SKLSDE-2014ZX-06.

References

[1] A. W. Williams and R. L. Probert, "A Measure for Component Interaction Test Coverage", ACS/IEEE International Conference on Computer Systems and Applications; Beirut, Lebanon, **(2001)**.
[2] M. Grindal, J. Offutt and S. F. Andler, "Combination Testing Strategies: A Survey. Softw. Test", Verif. Rel., vol.15, no.3, **(2005)**.
[3] C. Yilmaz, M. B. Cohen and A. A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces", IEEE Trans. Software Eng., vol.32, no.1, **(2006)**.
[4] I. S. Dunietz, W. K. Ehrlich and B. D. Szablak, "Applying Design of Experiments to Software Testing: Experience Report", Proceedings of the 19th international conference on Software engineering; Japan, **(1997)** May 17-23.
[5] D. R. Kuhn, D. R. Wallace and AM Gallo, "J. Software Fault Interactions and Implications for Software Testing", IEEE Trans. Software Eng., vol. 30, no.6, **(2004)**.
[6] M. B. Cohen, P. B. Gibbons and W. B. Mugridge, "A Variable Strength Interaction Testing of Components", Proceedings of 27th International Conference on Computer Software and Applications; Dallas, Texas, **(2003)** November 3-6.
[7] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing", Software Engineering Workshop, Proceedings of 27th Annual NASA Goddard/IEEE; USA, **(2002)** December 5-6.
[8] M. B. Cohen, P. B. Gibbons and W. B. Mugridge, "Constructing Test Suites for Interaction Testing", Proceedings of 25th International Conference on Software Engineering; Portland, Oregon, **(2003)** May 3-10.

[9] Y. Lei, R. Kacker and D. R. Kuhn, "IPOG: A General Strategy for t-way Software Testing", Proceedings of 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems; Tucson, AZ, **(2007)**: March 26-29.
[10] P. Nayeri, C. J. Colbourn and G. Konjevod, "Randomized Post-optimization of Covering Arrays", Combinatorial Algorithms. Springer Berlin Heidelberg, **(2009)**.
[11] P. Nayeri, C. J. Colbourn and G. Konjevod, "Randomized Post-optimization of Covering Arrays", European Journal of Combinatorics, vol. 34, no.1, **(2013)**.
[12] C. H. Nie and H. Leung, "A Survey of Combinatorial Testing", ACM Comput. Surv., vol.43, no.2, **(2011)**.
[13] D. M. Cohen, S. R. Dalal and M. L Fredman, "The AETG system: An Approach to Testing Based on Combinatorial Design", IEEE Trans. Software Eng., vol.23, no.7, **(1997)**.
[14] R. C. Bryce and C. J. Colbourn, "The Density Algorithm for Pairwise Interaction Testing", Softw. Test. Verif. Rel., vol.17, no.3, **(2007)**.
[15] R. C. Bryce and C. J. Colbourn, "A Density‐based Greedy Algorithm for Higher Strength Covering Arrays", Softw. Test. Verif. Rel., vol.19, no.1, **(2009)**.
[16] K. C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing", IEEE Trans. Software Eng., vol.28, no.1, **(2002)**.
[17] Y. Lei, R. Kacker and D. R. Kuhn, "IPOG/IPOG‐D: Efficient Test Generation for Multi-way Combinatorial Testing", Softw. Test. Verif. Rel., vol.18, no.3, **(2008)**.
[18] M. Forbes, J. Lawrence and Y. Lei, "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays", Journal of Research of the National Institute of Standards and Technology, vol.113, no.5, **(2008)**.
[19] S. Gao, J. Lv, B. Du, Y. Jiang and S. Ma, "General Optimization Strategies for Refining the In-Parameter-Order Algorithm", Proceedings of IEEE 14th International Conference on Quality Software; Dallas, Texas, **(2014)** October 2-3.
[20] S. Gao, J. Lv, B. Du, C. J. Charles and S. Ma, "Balancing Frequencies and Fault Detection in the In-Parameter-Order Algorithm", J. Comput. Sci. & Technol., vol.30, no.5, **(2015)**.
[21] S. Gao, B. Du, Y. Jiang, J. Lv and S. Ma, "An Efficient Algorithm for Pairwise Test Case Generation in Presence of Constraints", Proceedings of IEEE 2nd International Conference on Systems and Informatics; Shanghai, China, **(2014)** November 15-17.
[22] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker and D. R. Kuhn, "An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation", Proceedings of IEEE 6th International Conference on Software Testing, Verification and Validation; Luembourg, **(2013)** March 18-22.
[23] B. Chen and J. Zhang, "Tuple Density: A New Metric for Combinatorial Test Suites (nier track)", Proceedings of the 33rd International Conference on Software Engineering; Waikiki, Hawaii, **(2011)** May 21-28.
[24] A. Hartman and L. Raskin, "Problems and Algorithms for Covering Arrays", Discrete Mathematics, vol. 284, no.1, **(2004)**.
[25] D. R. Kuhn, R. N. Kacker and Y. Lei, "Practical Combinatorial Testing. NIST Special Publication", vol.800, no.142, **(2010)**.
[26] J. R. Maximoff, M. D. Trela and D. R. Kuhn, "A Method for Analyzing System State-space Coverage within a t-wise Testing Framework", Proceedings of 4th Annual IEEE Systems Conference; San Diego, CA, **(2010)** April 5-8.
[27] D.R. Kuhn, R. Kacker and Y. Lei, "Combinatorial Coverage Measurement", NIST IR 7878. http://dx.doi.org/10.6028/NIST.IR.7878, **(2012)**
[28] D. R. Kuhn, I. D. Mendoza and R. N. Kacker, "Combinatorial Coverage Measurement Concepts and Applications", Proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW); Luxembourg, **(2013)** March 18-22.

Authors

Shiwei Gao, he received the M.S. degree from Yanshan University, China, in 2010. He is currently a Ph.D. candidate in School of Computer Science and Engineering of Beihang University, Beijing, China. His research interests include software testing, software reliability theory and automation control theory.

Jianghua Lv, he received her Ph.D. degrees in computer science from Jilin University, Changchun, in 2003. Currently she is an assistant professor in the School of Computer Science and Engineering of Beihang University, Beijing. Her research interests include software testing, software reliability theory and automation control theory.

Fan Yang, he received the B.S. degree from Beihang University, China, in 2015. He is currently working towards the Master in School of Computer Science and Engineering of Beihang University. His research interest is software testing.

Jiaqi Xie, he received the B.S. degree from Beihang University, China, in 2014. He is currently working towards the Master in School of Computer Science and Engineering of Beihang University. His research interest is software testing.

Shilong Ma, he is currently a professor and doctor tutor of the School of Computer Science and Engineering of Beihang University, Beijing. His main research focus is on computation models in networks, logic reasoning and behaviors in network computing.