Gao SW, Lv JH, Du BL *et al.* Balancing frequencies and fault detection in the in-parameter-order algorithm. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(5): 957–968 Sept. 2015. DOI 10.1007/s11390-015-1574-6

Balancing Frequencies and Fault Detection in the In-Parameter-Order Algorithm

Shi-Wei Gao [1] (高世伟), Jiang-Hua Lv [1,*] (吕江花), Bing-Lei Du [1] (杜冰磊), Charles J. Colbourn [2] and Shi-Long Ma [1] (马世龙)

[1] *State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China*
[2] *School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe AZ 85287-8809, U.S.A.*

E-mail: ge89@163.com; jhlv@nlsde.buaa.edu.cn; binglei.du@gmail.com; Charles.Colbourn@asu.edu slma@nlsde.buaa.edu.cn

Received March 16, 2015; revised June 29, 2015.

Abstract The In-Parameter-Order (IPO) algorithm is a widely used strategy for the construction of software test suites for combinatorial testing (CT) whose goal is to reveal faults triggered by interactions among parameters. Variants of IPO have been shown to produce test suites within reasonable amounts of time that are often not much larger than the smallest test suites known. When an entire test suite is executed, all faults that arise from t-way interactions for some fixed t are surely found. However, when tests are executed one at a time, it is desirable to detect a fault as early as possible so that it can be repaired. The basic IPO strategies of horizontal and vertical growth address test suite size, but not the early detection of faults. In this paper, the growth strategies in IPO are modified to attempt to evenly distribute the values of each parameter across the tests. Together with a reordering strategy that we add, this modification to IPO improves the rate of fault detection dramatically (improved by 31% on average). Moreover, our modifications always reduce generation time (2 times faster on average) and in some cases also reduce test suite size.

Keywords combinatorial testing, IPO, test suite generation, expected time to fault detection, software under test

1 Introduction

Modern software systems are highly configurable. Their behavior is controlled by many parameters. Interactions among these parameters may cause severe failures, resulting in poor reliability. Therefore, software testing and reliability assessment are crucial in the design of effective software, as discussed in [1-3] for reliability and in [4-15] for software testing. Software testing serves two main purposes: 1) to ensure that software has as few errors as possible prior to release, and 2) to detect and isolate faults in the software. A generic model of such a software system identifies a finite set of parameters, and a finite set of possible values for each parameter. Faults may arise due to a choice of a value for a single parameter, interactions among the values of a subset of the parameters, or a result of environmental conditions not included in the software model. We focus on the faults that arise from the parameters identified and the interactions among them. It is nearly always impractical to exhaustively test all combinations of parameter values because of resource constraints. Fortunately, this is not necessary in general: in some real software systems, more than 70 percent of faults are caused by interactions between two parameters[16], and all known faults are caused by interactions among six or fewer parameters[17-18].

Regular Paper
Special Section on Software Systems
This work was supported by the National Natural Science Foundation of China under Grant Nos. 61300007 and 61305054, the Fundamental Research Funds for the Central Universities of China under Grant Nos. YWF-15-GJSYS-106 and YWF-14-JSJXY-007, and the Project of the State Key Laboratory of Software Development Environment of China under Grant Nos. SKLSDE-2015ZX-09 and SKLSDE-2014ZX-06.
*Corresponding Author

For these reasons, combinatorial testing (CT) or t-way testing chooses a strength t (the largest number of parameters interacting to cause a fault), and forms a software interaction test suite as follows. Every row of the test suite is a test or a test case. For each parameter in the system, each test specifies an admissible value for the parameter. The defining property is that, no matter how one chooses t parameters and an admissible value for each (a t-way interaction), at least one test has the specified parameters set to the indicated values. This coverage property ensures that every possible interaction among t or fewer parameter values must arise in at least one of the test cases. CT has proved to be an efficient testing technique for software[6,9,19]. Indeed, empirical studies have shown that t-way testing can effectively detect faults in various applications[17-18,20-22].

A primary objective in producing a test suite is to minimize the cost of executing the tests; hence minimizing the number of tests is desired. At the same time, however, the time to produce the test suite is also crucial. Hence the most effort has been invested in finding a variety of test suite generation algorithms. Some invest additional computational resources in minimizing the size of the test suite, while others focus on fast generation methods for test suites of acceptable but not minimum size. General methods providing fast generation have primarily involved greedy algorithms[9]. One-test-at-a-time methods start with an empty test suite, and keep track of the as-yet-uncovered t-way interactions. Then repeatedly a test is selected, which attempts to maximize the number of such interactions that are covered by the test, until all interactions are covered. This strategy was pioneered in AETG[23], and later proved to be within a constant factor of the optimal size[24-25]. In practice, maintaining a list of all t-way interactions can be prohibitive when the number of parameters is large. One-parameter-at-a-time methods instead construct a test suite for t of the parameters (this contains all of the possible tests). Then it repeatedly adds a new parameter, and chooses a value for this parameter in each of the existing tests (horizontal growth). Because it is possible that some t-way interactions involving the new parameter have not been covered yet, further tests are selected to cover all such interactions (vertical growth). This requires maintaining a list of $(t-1)$-way interactions, and hence can involve less bookkeeping. The pioneering example here is IPO[26] and its extensions, IPOG[27], and IPOG-F and IPOG-F2[28], which will be discussed in more detail in Section 2. Both strategies typically produce test suites of acceptable size[26,29]. It has been observed that one-test-at-a-time methods produce slightly smaller test suites in general, while one-parameter-at-a-time methods are somewhat faster at generation[26].

As mentioned earlier, software interaction test suites serve as two complementary roles[30]: to verify that no t-way interaction of SUT (software under test) causes a fault, or to locate such a fault. These two roles are different: certifying absence of a fault requires running the whole test suite, while locating a fault may not. Indeed in [30], it is shown that minimum test suite size is not the correct objective for fault location; the structure of the test suite can be more important than its size alone. An improved rate of fault detection can provide faster feedback to testers[31]. Recent studies have shown that CT is an effective fault detection technique and that early fault detection can be improved by reordering the generated test suites using interaction-based prioritization approaches[32-34]. Many strategies have been proposed to guide prioritization using evaluation measures such as interaction coverage based prioritization[30,35-39] and incremental interaction coverage based prioritization[40-41]. In [30], an evaluation measure of the expected time to fault detection is given.

Test case prioritization techniques have been explored for the one-test-at-a-time methods, but little is known for the one-parameter-at-a-time methods. Bryce *et al.*[35-36,42] presented techniques that combine generation and prioritization. Pure prioritization[32-34,39] instead reorders an existing interaction test suite, using the metric of normalized average percentage of faults detected (NAPFD). However, existing pure prioritization techniques use explicit fault measurements of real systems, and hence are not directly suitable for the IPO algorithm.

The main contributions of our work are:

1) We modify the IPO algorithm in order to accelerate the method and make it effective for fault detection. Our modifications attempt to make the values of each parameter more evenly distributed during generation. We focus on choosing values for the extension to an additional parameter during the horizontal growth of the algorithm and filling values for *don't care* positions. (See Section 3.)

2) We develop a pure prioritization technique (a reordering strategy) for the IPO algorithm based on the evaluation measure presented in [30]. Our method can reduce the expected time to fault detection effectively. (See Section 4.)

3) We conduct experiments to demonstrate the effectiveness of the modifications (see Section 5). We conclude that the modifications to the IPO strategy result in faster generation (2 times faster on average according to the experimental results in Subsection 5.1), sometimes in smaller test suites, and together with the pure prioritization, in less time to detect the first fault (improved by 31% on average according to the experimental results in Subsection 5.2).

2 Framework of the IPO Algorithm

IPO comprises a family of methods of the one-parameter-at-a-time type. We focus on IPOG as a representative implementation. The basic operation is to add a new parameter to an existing interaction test suite of strength t. To initialize the method, whenever the number of parameters is at most t, all possible rows are included, which is necessary and sufficient to obtain a test suite.

Thereafter, to introduce a new parameter, the set π of all t-way interactions involving the new parameter is computed. Horizontal growth adds a value of the new parameter to each existing row so that this extended row covers the most interactions in π; the interactions covered are removed from π. Then if π still contains uncovered interactions, vertical growth adds new rows to cover them. This process is outlined in the flowchart in Fig.1.

Existing variants of the IPO strategy alter the selection of values for the new parameter during horizontal growth and the selection of additional rows during vertical growth. During both horizontal and vertical growth, it frequently happens that the value for one or more parameters in a row can be chosen arbitrarily without affecting the coverage of the row. Such entries are *don't care* positions[26] in the test suite. The IPO methods exploit the fact that selecting values for *don't care* positions can be deferred; then they can be filled during horizontal growth when the next parameter is introduced. Every variant of IPO must therefore deal with two basic problems:

- choose values for the new parameter to maximize the number of uncovered interactions covered during horizontal growth;
- assign values for *don't care* positions that arise.

In the next section, we explore an implementation of this IPO framework in which the objective is not just to ensure coverage, but also to attempt to make each value appear as equally often as possible for each parameter. The latter is a *balance* condition.

3 Balance in the IPO Algorithm

A test suite must cover all t-way interactions. Consider a specific parameter and the t-way interactions that contain it. For each value of the parameter, the numbers of these t-way interactions with each different value of the parameter are the same. Now consider the frequencies of values of the parameter within the tests of a test suite. Because each value must provide the coverage of the same number of interactions, it appears to be reasonable to attempt to make the frequencies

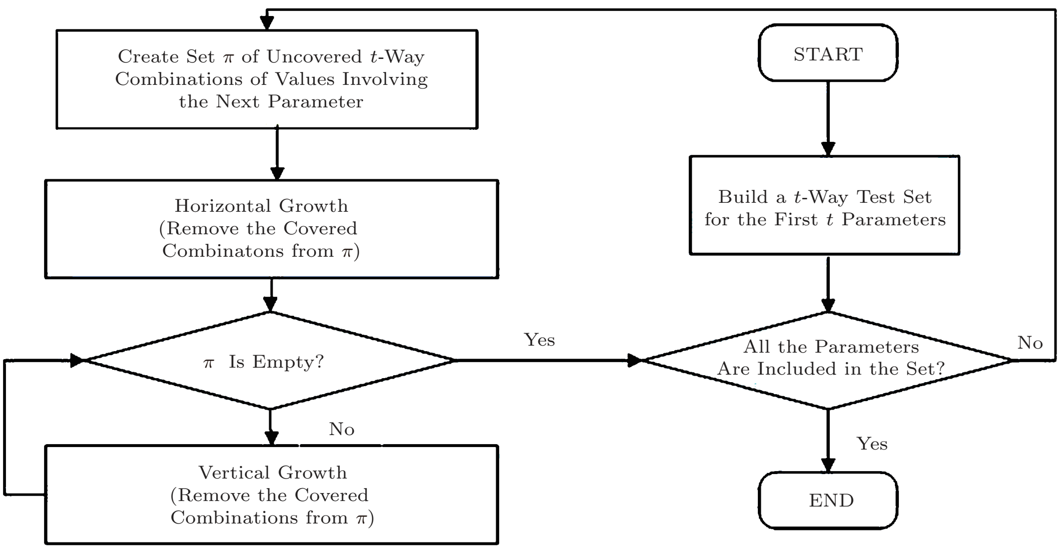


Fig.1. Flowchart of IPOG algorithm.

close to equal. The same argument applies to fault detection.

Two issues arise. First, current IPO algorithms do not make any explicit effort to balance the frequencies of values. Second, it is not at all clear how such an objective might affect the sizes of test suites produced, or the time to generate them, or their rates of fault detection. In this section, we develop modifications of IPO to address frequencies of values. Subsequent sections treat their impacts.

3.1 Choosing a New Parameter's Values

During horizontal growth, the IPOG algorithm chooses to add a value of the new parameter to cover the greatest number of interactions in π. In many situations, more than one value achieves this goal, and we must choose one. A naive strategy treats the values as ordered, and selects the smallest value that covers the most interactions in π. This introduces a bias towards the smaller values of each parameter, sometimes resulting in smaller values appearing much more frequently than larger ones.

Here a different strategy, shown in Algorithm 1, is proposed. The essential change is to treat the values as being cyclically ordered, recording the value selected for the previous row. Then possible values for this row are considered by starting from the value following the previous one selected. For this modification, vertical growth remains unchanged.

Algorithm 1. Modified Horizontal Growth

$Cov[r; v]$ is the number of interactions that the extended row $(r; v)$ covers
$q \leftarrow |P|$
$prev \leftarrow q$
for each row r in the covering array ca **do**
    $max \leftarrow (prev + 1)$ **mod** q
    $j \leftarrow (max + 1)$ **mod** q
    while $j \neq ((prev + 1)$ **mod** $q)$ **do**
        if $Cov[r, v_j] > Cov[r, v_{max}]$ **then**
            $max \leftarrow j$
        end if
        $j \leftarrow (j + 1)$ **mod** q
    end while
    $r \leftarrow (r, v_{max})$
    $prev \leftarrow max$
end for

Algorithm 1 incurs additional time to track the previous value selected, but this small addition is dominated by the computation of coverage, and hence makes no change in the complexity of the method.

While shown in Algorithm 1 for IPOG, this simple strategy can also be used in IPOG-F and IPOG-F2. We show the modification for IPOG-F. The IPOG-F algorithm greedily selects over both the row and the value with which the covering array is extended, and the extended row/value pair $(i; a)$ is greedily selected by the following formula[28]:

$$t_n = \binom{n-1}{t-1} - T_c[i; a],$$

where n is the number of parameters, $T_c[i; a]$ denotes the t-tuples that have previously been covered by already extended rows, and t_n denotes the number of new t-tuples the row/value pair would cover if we extend row i with value a. The metric of optimal selection for the extended row $(i; a)$ is that the extended row $(i; a)$ would maximize t_n.

The original pseudo-code for horizontal growth in IPOG-F is shown in Algorithm 2. The modification replaces line 6 to line 10 of Algorithm 2 as shown in Algorithm 3. Similar modifications can be applied to IPOG-F2.

Algorithm 2. Horizontal Growth of IPOG-F

1. $T_c[r; a]$ is the number of t-tuples covered by $(r; a)$
2. $Cov[\Lambda, v]$ is `true` if the interaction with column tuple Λ and value tuple v is covered `false` otherwise
3. $T_c[i; a] \leftarrow 0, \forall i, a$
4. $Cov[\Lambda, v] \leftarrow$ `false`$, \forall \Lambda, a$
5. **while** some row is non-extended **do**
6.   Find non-extended row i and value a
7.   so that $t_n = \binom{k-1}{t-1} - T_c[i; a]$ is maximum
8.   **if** $t_n = 0$ **then**
9.   Stop horizontal growth
10.   **end if**
11.   Extend row i with value a
12.   **for all** non-extended row j **do**
13. $S \leftarrow$ set of columns where rows i and j have identical entries
14.   **for all** column tuples $\Lambda \subset S$ **do**
15. $v \leftarrow$ the value tuple in row i and column tuple Λ
16.   **if** $Cov[\Lambda, v] =$ `false` **then**
17. $T_c[j; a] \leftarrow T_c[j; a] + 1$
18.   **end if**
19.   **end for**
20.   **end for**
21.   **for all** column tuples Λ **do**
22. $v \leftarrow$ the value tuple in row r and column tuple Λ
23.   **if** $Cov[\Lambda, v] =$ `false` **then**
24. $Cov[\Lambda, v] \leftarrow$ `true`
25.   **end if**
26.   **end for**
27. **end while**

Algorithm 3. Modification (Lines 6∼10)

$max \leftarrow (prev + 1)$ **mod** q
$j \leftarrow (max + 1)$ **mod** q
while $j \neq ((prev + 1)$ **mod** $q)$ **do**
    if $T_c[i; v_j] < T_c[i, v_{max}]$ **then**
        $max \leftarrow j$
        $j \leftarrow (j + 1)$ **mod** q
    end if
end while
$a \leftarrow v_{max}$
$t_n \leftarrow \binom{k-1}{t-1} - T_c[\tau, a]$
if $t_n = 0$ **then**
    Stop horizontal growth
end if
Extend row i with value a
$prev \leftarrow max$

3.2 Addressing *don't care* Positions

In horizontal growth, when the maximum number of interactions that the extended row $(r; v)$ can cover is 0, the value at this position is a *don't care*. The *don't care* positions can be addressed using the method of Subsection 3.1.

In vertical growth, new rows that are created to cover the t-way combinations in π not covered by horizontal growth can leave positions not needed to cover interactions in π as *don't care*. The selection of these values can influence the extension for the remaining parameters. To exploit these *don't care* positions, one strategy focuses on coverage, and the other on balance.

The balance strategy attempts to make values of all parameters distributed evenly: as each *don't care* arises, it is filled with a value for this parameter that currently appears the least often; ties are handled by taking the next in the cyclic order of values after the previous selection.

The coverage strategy is greedy. *Don't care* positions produced in vertical growth are left unassigned until the next horizontal growth. Then a value is chosen so that the row covers the most uncovered interactions, using the method described in Subsection 3.1.

Focusing on coverage is generally slightly superior in reducing the size of test suites. However, the balance strategy reduces the time to generate the test suite. Because of our interest in fault detection, and the fact that existing IPO variants use a coverage strategy, we adopt the balance strategy here. The pseudo-code for the balance strategy is shown in Algorithm 4.

Vertical growth treating *don't care* positions using a coverage strategy examines all t-way interactions, while

Algorithm 4. Addressing *don't care* Positions

Number the values of P_i as $v_1, v_2, \ldots, v_{|P_i|}$
$freq[P_i, j]$ is the frequency of value v_j of P_i appears in the existing test set
e is an entry in column i
if e is a *don't care* position **then**
    Find min that $freq[P_i, min]$ is minimum in $freq[P_i, 1], \ldots, freq[P_i, |P_i|]$
    Assign e with v_{min}
end if

our balance strategy only examines frequencies. Savings are only incurred with the balance strategy when *don't care* positions arise during vertical growth. In both cases, the worst-case complexity is dominated by the cost of horizontal growth, so in principle the two methods have the same asymptotic complexity. However, in practice, every *don't care* position results in a saving in computation time for the balance strategy.

4 Reducing the Expected Time to Fault Detection

In [30], a measurement of the goodness of a test suite at detecting a fault is defined. Suppose that every test takes the same time to run. Further suppose that faults are randomly distributed among the t-way interactions, and that there is no *a priori* information about their location. For a system with s faults, the expected time to fault detection is determined by the expected number of tests to detect the presence of a fault. Φ_s denotes the expected number of tests to detect the first fault in a system with s faults.

$$\Phi_s = \frac{\sum_{i=1}^{N} \binom{u_i}{s}}{\binom{\Lambda}{s}}.$$

Here u_i is the number of uncovered interactions before executing the i-th row, N is the number of rows of the test suite, and Λ is the total number of t-way interactions.

This measure applies to any test suite when faults arise randomly, and is not intended to examine particular patterns of faults in specific systems. As such, it can serve as a means to evaluate test suites for use in an as-yet-unknown application.

Minimizing the expected time to fault detection means constructing a test suite to minimize Φ_s given s. Rather than constructing a test suite to minimize Φ_s directly, we can reorder the rows of a test suite to reduce Φ_s.

Because all faults of interest are caused by parameter interactions, the more uncovered interactions contained in the test, the more likely a fault is to be revealed. Hence placing the tests that cover the greatest number of the uncovered interactions early can increase the probability of detecting a fault. To see this, we rewrite the formula as follows:

$$\Phi_s = \frac{\sum_{i=1}^{N} \binom{u_i}{s}}{\binom{\Lambda}{s}} = \sum_{i=1}^{N} \frac{\binom{u_i}{s}}{\binom{\Lambda}{s}}.$$

Then the problem becomes minimizing the average value of $\frac{\binom{u_i}{s}}{\binom{\Lambda}{s}}$, the likelihood that all faults remain undetected after running i tests. The method for reordering the test suite is Algorithm 5. There may be a tie for row r_j where $T_c[r_j]$ is the largest — if there is, the tie would be broken randomly.

Algorithm 5. Reordering Test Suites

```
n ← N
for j from 1 to n do
    for each row r_1, ..., r_n
        Determine the number T_c[r_i] of t-way
        interactions covered in r_i
        but not covered in r_1, ..., r_{i-1}
    end for
    Choose a row r_j from r_i, ..., r_n for which
    T_c[r_j] is the largest
    if T_c[r_j] = 0 then
        Remove all rows r_i, ..., r_n from the suite
        n ← i − 1
    else
        Swap r_i and r_j in the suite
    end if
end for
```

5 Experiments

We employ the tool ACTS-2.8 (Advanced Combinatorial Testing System)[43], including implementations of IPOG, IPOG-F and IPOG-F2, etc. We compare the tool ACTS-2.8 with our variants of IPOG, IPOG-F and IPOG-F2 in which the handling of *don't care* positions attempts to balance frequencies of values; our versions are coded in C++. All of the experimental results reported here are performed on a laptop with Core™ 2 Duo Intel® processor clocked at 2.60 GHz and 4 GB memory.

5.1 Test Suite Size and Execution Time

First we examine the relative performance for different numbers of values for the parameters. The notation d^t indicates that there are t parameters, each with d values. To start, we vary the number of values. Table 1 shows execution time and test suite sizes when the strength is 4, and there are five parameters whose number of values is 5, 10, 15, or 20. As expected, the execution time for our methods is substantially smaller (see Fig.2). What is more surprising is that our methods consistently produce test suites no larger than the original methods, and sometimes produce much smaller ones.

Now we vary the number of parameters. Table 2 shows results when the strength is 4, the number of parameters is 10, 15, 20, or 25, and the number of values is 5. Again the execution time for our methods shows improvements (see Fig.3). However, as the number of parameters increases, the deferral in filling *don't care* positions by the original methods generally produces smaller test suite sizes.

Now we vary the strength. Table 3 presents results for 10^6 when the strength is 2, 3, 4, or 5. Once again, the execution time for our methods is substantially lower (see Fig.4). Our methods do not fare as well with respect to test suite size, but appear to be very effective when the strength is larger.

Our methods appear to improve execution time consistently as expected. Nevertheless, they also improve on test suite sizes in some cases, especially when the strength is large or the number of values is large. Real systems rarely have the same number of values for each parameter, so we also consider situations in which different parameters can have different numbers of values.

Table 4 presents results with strength 4 for five different sets of numbers of values for 10 parameters. Execution time improvements again arise for our algorithms. Moreover, a pattern for test suite sizes is clear: our methods improve when there is more variation in numbers of values.

Next we examine the relative performance using the Traffic Collision Avoidance System (TCAS), which has been utilized in several other studies of software testing[27,44-46]. TCAS has 12 parameters: seven parameters have two values, two parameters have three values, one parameter has four values, and two parameters have 10 values. Table 5 gives the results. (In [46], similar results for the original IPOG versions are given for the TCAS system.) While our improvements in execution time are evident, no obvious pattern indicates which method produces the smallest test suite.

Our methods have simplified the manner in which *don't care* positions are treated in order to balance the frequencies of values. Our experimental results all con-

Table 1. Results for Five Parameters with 5 to 20 Values for 4-Way Testing

Parameter	Our IPOG		IPOG(ACTS)		Our IPOG-F		IPOG-F(ACTS)		Our IPOG-F2		IPOG-F2(ACTS)	
Config.	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)
5^5	745	0.001	790	0.015	625	0.000	625	0.047	625	0.000	788	0.031
10^5	11 990	0.078	12 298	0.827	10 000	0.673	10 000	6.109	10 000	0.500	12 394	4.859
15^5	58 410	1.101	61 945	16.329	50 625	18.469	50 625	146.730	50 625	12.782	61 615	184.450
20^5	184 680	9.666	191 652	120.220	160 000	200.020	160 000	1 376.000	160 000	209.290	192 082	1 966.200

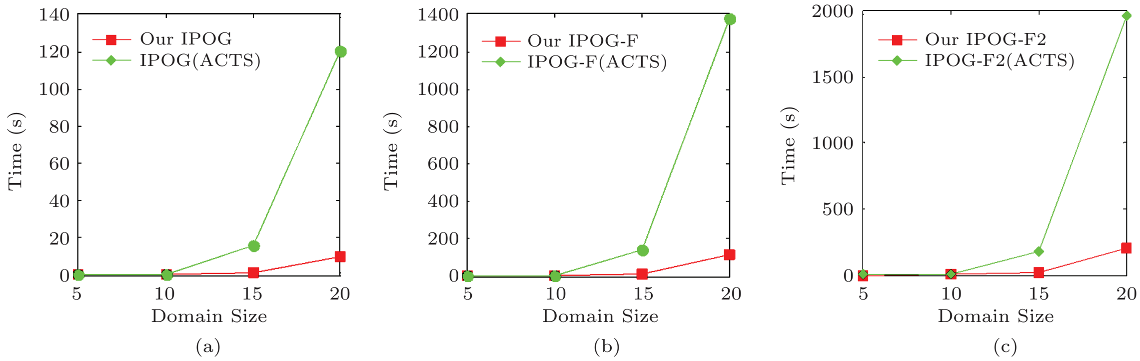


Fig.2. Execution time, varying the number of values (4-way). (a) IPOG. (b) IPOG-F. (c) IPOG-F2.

Table 2. Results for 10 to 25 5-Value Parameters for 4-Way Testing

Parameter	Our IPOG		IPOG(ACTS)		Our IPOG-F		IPOG-F(ACTS)		Our IPOG-F2		IPOG-F2(ACTS)	
Config.	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)
5^{10}	1 890	0.056	1 859	0.188	1 833	0.625	1 882	1.750	1 965	0.187	1 905	0.297
5^{15}	2 584	0.517	2 534	0.954	2 461	7.109	2 454	14.579	2 736	1.282	2 644	1.421
5^{20}	3 114	2.140	3 032	4.094	2 951	34.361	2 898	60.987	3 308	4.329	3 180	4.344
5^{25}	3 540	7.012	3 434	16.049	3 338	111.150	3 279	176.340	3 763	8.752	3 589	9.188

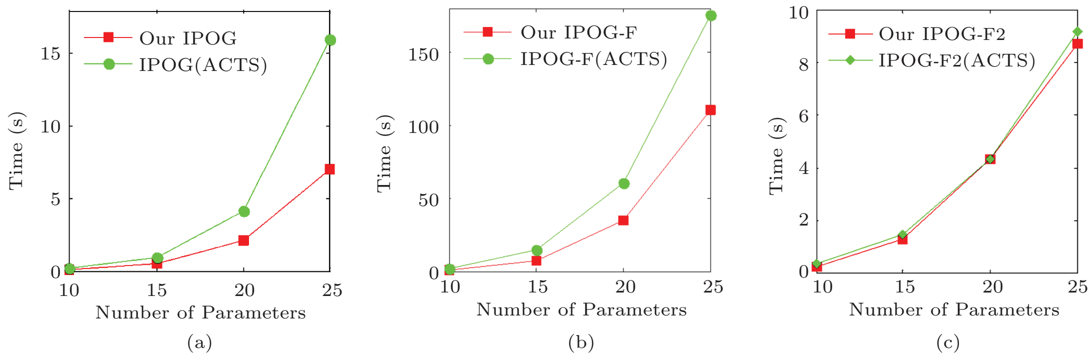


Fig.3. Execution time, increasing the number of parameters (4-way). (a) IPOG. (b) IPOG-F. (c) IPOG-F2.

Table 3. Results for Six 10-Value Parameters for 2∼5-Way Testing

t	Our IPOG		IPOG(ACTS)		Our IPOG-F		IPOG-F(ACTS)		Our IPOG-F2		IPOG-F2(ACTS)	
	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)
2	149	0.000	130	0.005	133	0.000	134	0.031	135	0.000	134	0.016
3	1 633	0.010	1 633	0.059	1 577	0.047	1 553	0.266	1 629	0.032	1 625	0.140
4	16 293	0.195	16 496	4.276	15 594	2.704	15 467	18.126	15 631	1.594	16 347	9.297
5	123 060	5.139	130 728	116.470	100 000	88.692	100 000	575.150	100 000	54.971	132 428	449.330

Table 4. Results for Five Systems with Different Numbers of Values in 4-Way Testing

Parameter Config.	Our IPOG		IPOG(ACTS)		Our IPOG-F		IPOG-F(ACTS)		Our IPOG-F2		IPOG-F2(ACTS)	
	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)
10^{10}	29 915	1.942	29 466	28.040	28 437	129.57	28 079	359.17	31 744	45.237	30 986	53.440
$10^5 9^5$	23 878	1.295	23 961	14.583	22 521	94.27	22 726	248.04	25 222	31.611	24 741	39.736
$15^3 10^4 5^3$	41 128	1.734	45 128	13.689	41 505	236.87	43 306	757.68	46 509	162.850	48 295	262.170
$16^1 15^2 10^4 5^2 4^1$	42 913	1.750	47 591	14.532	43 774	249.37	45 693	289.72	48 660	148.510	51 147	149.510
$17^1 16^1 15^1 10^4 5^1 4^2$	47 248	1.844	52 991	14.860	48 847	235.95	50 287	333.89	54 099	189.290	57 634	199.810

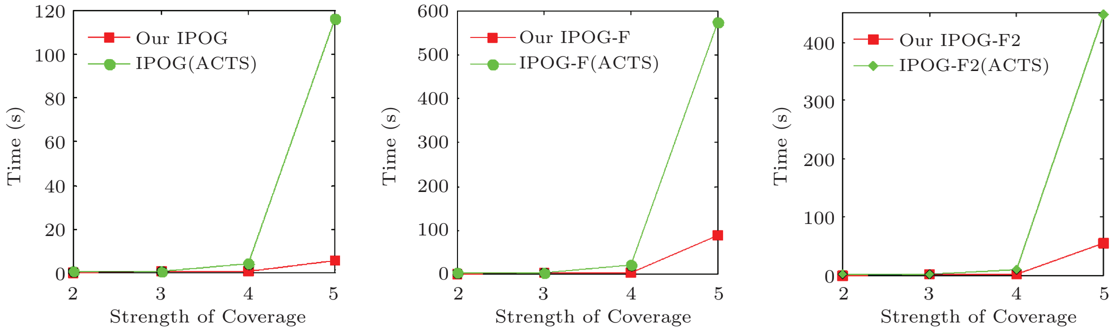


Fig.4. Execution time, increasing the test strength. (a) IPOG. (b) IPOG-F. (c) IPOG-F2.

Table 5. Results for TCAS

t	Our IPOG		IPOG(ACTS)		Our IPOG-F		IPOG-F(ACTS)		Our IPOG-F2		IPOG-F2(ACTS)	
	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)	Size	Time (s)
2	100	0.001	100	0.002	100	0.002	100	0.015	100	0.004	100	0.017
3	404	0.009	400	0.007	400	0.025	402	0.087	431	0.044	438	0.061
4	1 306	0.065	1 359	0.031	1 269	0.323	1 349	1.117	1 639	0.489	1 653	0.572
5	4 464	0.411	4 233	0.219	4 068	4.104	4 245	13.405	5 129	4.133	5 034	4.379
6	11 774	1.463	11 021	3.233	11 381	32.870	11 257	101.330	13 323	18.030	13 379	20.959

firm that this can dramatically reduce the execution time. One might have expected a substantial degradation in the test suite sizes produced. However, our results indicate not only that the balancing strategy is competitive, but also that it can improve test suite sizes.

Fast methods such as IPO do not generally produce the smallest test suites possible. To illustrate this, we apply a post-optimization method from [47-48] to some of the TCAS results. For strength 4, we treat the solutions for IPOG-F2; within 10 minutes of computation, post-optimization reduces the solution by our method from 1 639 to 1 201 rows, and the solution by the original method from 1 653 to 1 205 rows. For strength 5, we treat the solutions for IPOG-F; within one hour of computation, post-optimization reduces the solution by our method from 4 068 to 3 600 rows, and the solution by the original method from 4 245 also to 3 600 rows. For strength 6, we treat the solutions for IPOG; within 10 hours, post-optimization reduces the solution by our method from 11 774 to 9 794 rows, and the solution by the original method from 11 021 to 9 798 rows. By contrast, in a comparison of six different one-parameter-at-a-time methods[46], the best result has 10 851 rows. While the test suites from one-parameter-at-a-time methods are therefore definitely not the smallest, post-optimization is much more time-consuming and it requires a test suite as input. As the number of parameters increases, the speed with which an initial test suite can be constructed is crucial.

5.2 Expected Time to Fault Detection

Accelerating the IPO methods, even with a possible loss of accuracy in test suite size, can be worthwhile. However, a second concern is with potential performance in revealing faults. We examine the TCAS system, using our and the original versions of the three IPO variants. We examine the time to find the first fault when 1, 2, or 3 faults are present and when the strength is between 2 and 6. In our model, the time to execute each test is the same, so the expected time is directly proportional to the expected number of tests or rows needed. We consider test suites before and after our reordering.

Table 6 gives the results. To assess the efficacy of our modifications, we report two lines for each method and each strength; the first reports results for our methods, and the second for the original methods. Φ_1, Φ_2 and Φ_3 denote the expected number of tests to detect the first fault when there are one, two or three faults that are randomly chosen.

These results indicate that reordering is effective in reducing the time to fault detection, both for our methods and for the original ones. Fig.5 shows Φ_2 for each strength before and after the reordering for our methods, showing a substantial reduction from reordering. Fig.6 instead shows the expected number of tests when zero, one, two, or three faults are present. It appears that the reordering method is the most effective when the number of faults is small. This should be expected, because the presence of many faults ensures that one will be found early no matter what ordering is used.

Our methods, despite often producing larger test suites, fare well with respect to expected time to fault detection. Comparing the performance of ours and the original IPOG when $t = 6$, for example, although our test suite is larger, it would yield smaller expected time to detect faults once reordered. Evidently the size of the

Table 6. Expected Time to Fault Detection for TCAS Before and After Reordering

Algorithm	t	Number of Faults					
		Φ_1		Φ_2		Φ_3	
		Before	After	Before	After	Before	After
IPOG	2	24.80	19.65	10.72	9.26	6.27	5.83
		24.81	19.65	10.73	9.26	6.27	5.85
	3	117.90	82.15	53.30	38.57	30.08	23.69
		117.68	82.12	53.18	38.47	30.03	23.56
	4	407.20	275.38	200.45	131.93	118.33	82.38
		408.42	276.34	201.60	132.73	119.08	82.77
	5	1 348.26	850.74	707.82	421.49	436.03	268.03
		1 348.14	848.20	708.24	421.30	436.40	268.37
	6	3 015.32	2 127.94	1 682.31	1 095.54	1 097.27	719.43
		3 007.69	2 140.04	1 680.95	1 106.03	1 096.56	725.45
IPOG-F	2	28.36	20.43	12.73	9.58	7.29	6.01
		27.19	20.67	12.18	9.67	7.08	6.02
	3	120.96	81.47	55.81	38.33	31.84	23.71
		120.94	81.34	55.99	38.18	32.13	23.72
	4	411.37	272.86	204.59	132.18	121.66	82.83
		411.97	269.36	204.73	129.68	121.75	81.77
	5	1 353.42	828.83	716.08	411.92	444.08	263.23
		1 354.28	822.73	715.52	410.22	443.26	261.36
	6	3 076.17	2 090.57	1 722.82	1 065.49	1 129.80	698.08
		3 017.29	2 059.33	1 693.94	1 063.99	1 109.05	700.68
IPOG-F2	2	26.44	20.52	11.90	9.75	6.98	6.13
		26.27	20.19	11.67	9.56	6.80	6.00
	3	120.61	82.75	55.26	38.36	31.49	23.84
		121.04	81.63	55.61	38.15	31.76	23.55
	4	419.07	275.05	207.11	130.39	123.20	81.95
		421.75	278.10	208.20	131.82	123.79	82.43
	5	1 378.15	844.54	724.94	412.79	449.34	263.65
		1 377.84	838.77	725.02	409.71	449.50	261.35
	6	3 129.21	2 127.62	1 732.44	1 068.73	1 133.78	699.08
		3 138.02	2 121.97	1 736.29	1 062.64	1 136.22	695.18

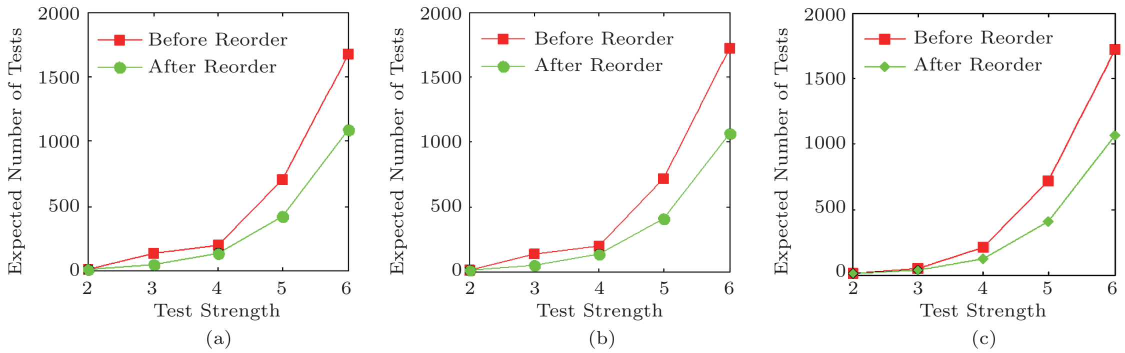


Fig.5. Expected number of tests for Φ_2. (a) IPOG. (b) IPOG-F. (c) IPOG-F2.

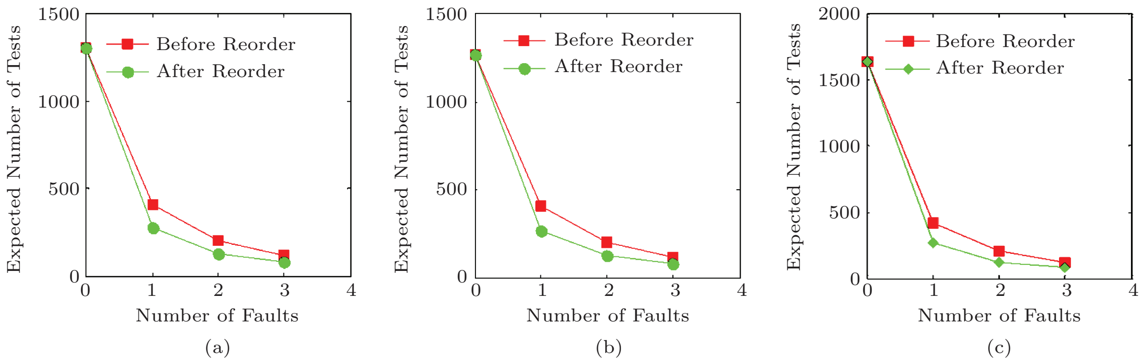


Fig.6. Expected number of tests, increasing the number of faults in TCAS (4-way). (a) IPOG. (b) IPOG-F. (c) IPOG-F2.

test suite, while relevant, is not the only factor affecting the expected time. Our results suggest that faster IPO implementations remain competitive, and hence that the objective of balancing frequencies of values is a reasonable one to pursue.

6 Conclusions

We identified three main goals in generating a test suite: time to generate the test suite, time to execute the test suite (test suite size), and the rate of fault detection. Our methods focus on reducing the time for generation, without severe negative impact on test suite size and fault detection. We accelerated variants of the IPO method by simplifying the manner in which *don't care* positions are filled. This results in a consistent improvement in the execution time to construct a test suite, but sacrifices to some extent the algorithm's ability to exploit such positions in repeated horizontal growth phases. This is reflected in our experimental results. While in numerous cases, our modifications find smaller test suites, in the others they do not. This occurs particularly when the number of parameters is large.

Any method to fill *don't care* positions immediately would be expected to accelerate the methods; however we devised a simple method that strives to balance the frequency of values for each parameter. We argued that such an objective can result in more effective horizontal growth, and that it can permit us to retain effective rates of fault detection. Both of these motivations are borne out by the experimental data.

One-test-at-a-time generation methods explicitly aim for good rates of fault detection by covering interactions early in the test suite, while one-parameter-at-a-time methods like IPO do not. Nevertheless, we showed that a reordering strategy can be applied to make dramatic improvement on the rate of fault detection.

If test suite size is a primary objective, using our methods together with randomized post-optimization[47-48] appears to be worthwhile. If expected time to fault detection is paramount, extending reordering to discover and replace *don't care* positions appears to be viable. Both merit further study. We suggest that both can benefit from balancing frequencies of values, a fast and simple way to generate useful test suites.

References

[1] Birnbaum Z W. On the importance of different components in a multicomponent system. In *Multivariate Analysis*, Krishnaiah P R (ed.), New York: Academic Press, 1969, pp.591-592.

[2] Kuo W, Zhu X. Relations and generalizations of importance measures in reliability. *IEEE Trans. Rel.*, 2012, 61(3): 659-674.

[3] Kuo W, Zhu X. Some recent advances on importance measures in reliability. *IEEE Trans. Rel.*, 2012, 61(2): 344-360.

[4] Anand S, Burke E K, Chen T Y *et al.* An orchestrated survey of methodologies for automated software test case generation. *J. Sys. Software*, 2013, 86(8): 1978-2001.

[5] Chen T Y, Kuo F C, Liu H *et al.* Code coverage of adaptive random testing. *IEEE Trans. Rel.*, 2013, 62(1): 226-237.

[6] Grindal M, Offutt J, Andler S F. Combination testing strategies: A survey. *Softw. Test. Verif. Rel.*, 2005, 15(3): 167-199.

[7] Hao D, Zhang L M, Zhang L *et al.* A unified test-case prioritization approach. *ACM Trans. Soft. Eng. Method*, 2014, 24(2): 10:1-10:31.

[8] Harman M, McMinn P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Software Eng.*, 2010, 36(2): 226-247.

[9] Nie C H, Leung H. A survey of combinatorial testing. *ACM Comput. Surv.*, 2011, 43(2): 11:1-11:29.

[10] Nebut C, Fleurey F, Le Traon Y *et al.* Automatic test generation: A use case driven approach. *IEEE. Trans. Software Eng.*, 2006, 32(3): 140-155.

[11] Perrouin G, Oster S, Sen S *et al.* Pairwise testing for software product lines: Comparison of two approaches. *Software. Qual. J.*, 2012, 20(3/4): 605-643.

[12] Xie T, Zhang L, Xiao X *et al.* Cooperative software testing and analysis: Advances and challenges. *Journal of Computer Science and Technology*, 2014, 29(4): 713-723.

[13] Yoo S, Harman M. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Rel.*, 2012, 22(2): 67-120.

[14] Zhang D M, Xie T. Software analytics: Achievements and challenges. In *Proc. the 35th Int. Conf. Software Eng.*, May 2013, p.1487.

[15] Yu K, Lin M, Chen J *et al.* Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *J. Sys. Software*, 2012, 85(10): 2305-2317.

[16] Bryce R C, Colbourn C J. One-test-at-a-time heuristic search for interaction test suites. In *Proc. the 9th Annu. Conf. Genetic and Evolutionary Computation*, Jul. 2007, pp.1082-1089.

[17] Kuhn D R, Reilly M J. An investigation of the applicability of design of experiments to software testing. In *Proc. the 27th Annu. NASA Goddard Workshop on Software Eng.*, Dec. 2002, pp.91-95.

[18] Kuhn D R, Wallace D R, Gallo Jr J A. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 2004, 30(6): 418-421.

[19] Cohen M B, Dwyer M B, Shi J. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 2008, 34(5): 633-650.

[20] Lei Y, Kacker R, Kuhn D R *et al.* IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing. *Softw. Test. Verif. Rel.*, 2008, 18(3): 125-148.

[21] Tung Y W, Aldiwan W S. Automating test case generation for the new generation mission software system. In *Proc. IEEE Aerospace Con.*, March 2000, pp.431-437.

[22] Wallace D R, Kuhn D R. Failure modes in medical device software: An analysis of 15 years of recall data. *Int. J. Rel., Quality and Safety Eng.*, 2001, 8(4): 351-371.

[23] Cohen D M, Dalal S R, Kajla A *et al.* The automatic efficient tests generator (AETG) system. In *Proc. the 5th Int. Sympo. Software Rel. Eng.*, Nov. 1994, pp.303-309.

[24] Bryce R C, Colbourn C J. The density algorithm for pairwise interaction testing. *Softw. Test. Verif. Rel.*, 2007, 17(3): 159-182.

[25] Bryce R C, Colbourn C J. A density-based greedy algorithm for higher strength covering arrays. *Softw. Test. Verif. Rel.*, 2009, 19(1): 37-53.

[26] Lei Y, Tai K C. In-parameter-order: A test generation strategy for pairwise testing. In *Proc. the 3rd Int. Symp. High-Assurance Sys. Eng.*, Nov. 1998, pp.254-261.

[27] Lei Y, Kacker R, Kuhn D R *et al.* IPOG: A general strategy for *t*-way software testing. In *Proc. the 14th Annu. Int. Conf. Worshop. Eng. Computer-Based Sys.*, March 2007, pp.549-556.

[28] Forbes M, Lawrence J, Lei Y, Kacker R N, Kuhn D R. Refining the in-parameter-order strategy for constructing covering arrays. *Journal of Research of the National Institute of Standards and Technology*, 2008, 113(5): 287-297.

[29] Cohen M B, Gibbons P B, Mugridge W B *et al.* Constructing test cases for interaction testing. In *Proc. the 25th Int. Conf. Software Eng.*, May 2003, pp.38-48.

[30] Bryce R C, Colbourn C J. Expected time to detection of interaction faults. *J. Combin. Mathematics and Combin. Comput.*, 2013, 86: 87-110.

[31] Rothermel G, Untch R H, Chu C *et al.* Prioritizing test cases for regression testing. *IEEE Trans. Software Eng.*, 2001, 27(10): 929-948.

[32] Qu X, Cohen M B. A study in prioritization for higher strength combinatorial testing. In *Proc. the 6th Int. Con. Software Testing, Verification and Validation, the 2nd Int. Workshops on Combinatorial Testing*, March 2013, pp.285-294.

[33] Qu X. Configuration aware prioritization techniques in regression testing. In *Proc. the 31st Int. Conf. Software Engineering, Companion Volume*, May 2009, pp.375-378.

[34] Qu X, Cohen M B, Rothermel G. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proc. Int. Symp. Software Tesing and Analysis*, July 2008, pp.75-86.

[35] Bryce R C, Colbourn C J. Test prioritization for pairwise interaction coverage. In *Proc. the 1st Int. Workshop on Advances in Model-Based Testing*, May 2005.

[36] Bryce R C, Colbourn C J. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Inform. Software Tech.*, 2006, 48(10): 960-970.

[37] Huang R, Chen J, Li Z, Wang R, Lu Y. Adaptive random prioritization for interaction test suites. In *Proc. the 29th Symp. Appl. Comput.*, March 2014, pp.1058-1063.

[38] Petke J, Yoo S, Cohen M B, Harman M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proc. the 12th Joint Meeting on European Software Engineering Conf. and the ACM SIGSOFT Symp. the Foundations of Software Eng.* (*ESEC/FSE 2013*), August 2013, pp.26-36.

[39] Qu X, Cohen M B, Woolf K M. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *Proc. the 23rd Int. Conf. Software Maintenance*, Oct. 2007, pp.255-264.

[40] Huang R, Chen J, Zhang T, Wang R, Lu Y. Prioritizing variable-strength covering array. In *Proc. the 37th IEEE Annu. Computer Software and Applications Conf.*, July 2013, pp.502-511.

[41] Huang R, Xie X, Towey D, Chen T Y, Lu Y, Chen J. Prioritization of combinatorial test cases by incremental interaction coverage. *Int. J. Softw. Eng. Know.*, 2014, 23(10): 1427-1457.

[42] Bryce R C, Memon A M. Test suite prioritization by interaction coverage. In *Proc. Workshop on Domain Specific Approaches to Software Test Automation*, September 2007, pp.1-7.

[43] Lei Y, Kuhn D R. Advanced combinatorial testing suite (ACTS). http://csrc.nist.gov/groups/SNS/acts/index.html, Aug. 2015.

[44] Hutchins M, Foster H, Goradia T *et al.* Experiments of the effectiveness of dataflow and control-flow-based test adequacy criteria. In *Proc. the 16th Int. Conf. Software Eng.*, May 1994, pp.191-200.

[45] Kuhn D R, Okun V. Pseudo-exhaustive testing for software. In *Proc. the 30th Annu. IEEE/NASA Software Engineering Workshop*, April 2006, pp.153-158.

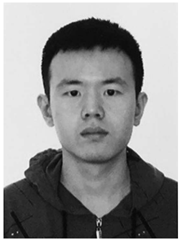
[46] Soh Z H C, Abdullah S A C, Zamil K Z. A distributed t-way test suite generation using "One-Parameter-at-a-Time" approach. *Int. J. Advance Soft Compu. Appl.*, 2013, 5(3): 91-103.

[47] Li X, Dong Z, Wu H *et al.* Refining a randomized post-optimization method for covering arrays. In *Proc. the 7th IEEE Int. Conf. Software Testing, Verification and Validation Workshops* (*ICSTW*), March 31-April 4, 2014, pp.143-152.

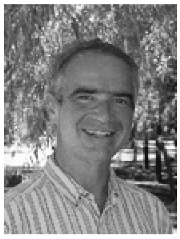
[48] Nayeri P, Colbourn C J, Konjevod G. Randomized post-optimization of covering arrays. *Eur. J. Combin.*, 2013, 34(1): 91-103.

Shi-Wei Gao received his B.S. degree in computer science and technology from Dezhou University, Dezhou, in 2007, and M.S. degree in information science and engineering from Yanshan University, Qinhuangdao, in 2010. He is currently a Ph.D. candidate in the School of Computer Science and Engineering of Beihang University, Beijing. He is a member of the State Key Laboratory of Software Development Environment of Beihang University. His research interests include software testing, software reliability theory, and formal methods.

Jiang-Hua Lv received her B.S. and Ph.D. degrees in computer science from Jilin University, Changchun, in 1998 and 2003, respectively. Currently she is an assistant professor in the School of Computer Science and Engineering of Beihang University, Beijing. She is a member of the State Key Laboratory of Software Development Environment of Beihang University. Her research focuses on formal theory and technology of software, theory and technology of testing, automatic testing of safety critical systems, and device collaboration.



Bing-Lei Du is currently an undergraduate in the School of Computer Science and Engineering of Beihang University, Beijing, and has been an intern in the State Key Laboratory of Software Development Environment of Beihang University since 2013. His research interest is software testing.



Charles J. Colbourn earned his Ph.D. degree in computer science from the University of Toronto in 1980, and is a professor of computer science and engineering at Arizona State University. He is the author of The Combinatorics of Network Reliability (Oxford), Triple Systems (Oxford), and 320 refereed journal papers focusing on combinatorial designs and graphs with applications in networking, computing, and communications. In 2004, he was awarded the Euler Medal for Lifetime Research Achievement by the Institute for Combinatorics and its Applications.

Shi-Long Ma is currently a professor and doctor tutor of the School of Computer Science and Engineering of Beihang University, Beijing. He is a member of the State Key Laboratory of Software Development Environment of Beihang University. His main research focus is on computation models in networks, logic reasoning and behaviors in network computing, and the theory of automatic testing.