SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2008; **18**:125–148
Published online 29 November 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.381

IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing

Yu Lei[1,*,†], Raghu Kacker[2], D. Richard Kuhn[2], Vadim Okun[2] and James Lawrence[3]

[1]*Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, TX 76019-0015, U.S.A.*
[2]*Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899-8910, U.S.A.*
[3]*Department of Mathematics, George Mason University, Fairfax, VA 22030, U.S.A.*

SUMMARY

This paper presents two strategies for multi-way testing (i.e. *t*-way testing with *t*>2). The first strategy generalizes an existing strategy, called in-parameter-order, from pairwise testing to multi-way testing. This strategy requires all multi-way combinations to be explicitly enumerated. When the number of multi-way combinations is large, however, explicit enumeration can be prohibitive in terms of both the space for storing these combinations and the time needed to enumerate them. To alleviate this problem, the second strategy combines the first strategy with a recursive construction procedure to reduce the number of multi-way combinations that have to be enumerated. Both strategies are deterministic, i.e. they always produce the same test set for the same system configuration. This paper reports a multi-way testing tool called FireEye, and provides an analytic and experimental evaluation of the two strategies. Copyright © 2007 John Wiley & Sons, Ltd.

Received 1 December 2006; Revised 5 October 2007; Accepted 11 October 2007

KEY WORDS: combinatorial testing; interaction testing; multi-way testing; software testing

*Correspondence to: Yu Lei, Department of Computer Science and Engineering, The University of Texas at Arlington, Box 19015, Arlington, TX 76019-0015, U.S.A.
†E-mail: ylei@cse.uta.edu

Contract/grant sponsor: Information Technology Lab (ITL) of the National Institute of Standards and Technology [NIST]; contract/grant number: 60NANB6D6192

1. INTRODUCTION

Combinatorial testing creates tests by selecting values for input parameters and by combining these values [1]. For a system with k parameters, each of which has v values, the number of possible combinations of values of these parameters is v^k. Owing to resource constraints, it is nearly always impractical to exhaustively test all possible combinations. Thus, a strategy is needed to select a subset of combinations to be tested. One such strategy, called t-way testing, requires every combination of values of *any* t parameters to be covered by at least one test, where t is referred to as the strength of coverage and usually takes a small value. Each combination of values of a set of parameters is considered to represent one possible interaction among these parameters. The rationale behind t-way testing is that not every parameter contributes to every fault, and many faults can be exposed by interactions involving only a few parameters. The notion of t-way testing can substantially reduce the number of tests. For example, a system of 20 parameters that have 10 values each requires 10^{20} tests for exhaustive testing, but as few as 180 tests for 2-way (or pairwise) testing [2]. Empirical studies have shown that t-way testing can effectively detect faults in various types of applications [3–6].

To illustrate the concept of t-way testing, consider an elementary software system consisting of three Boolean parameters. Denote the two values of a Boolean parameter by 0 and 1. Figure 1 shows a pairwise test set for this system. In this test set, each row represents a test, and each column represents a parameter (in the sense that each entry in a column is a value of the parameter represented by the column). An examination of this test set reveals that each of the three pairs of columns, i.e. columns 1 and 2, columns 1 and 3, and columns 2 and 3, contains all four pairs of values of two Boolean parameters, i.e. $\{00, 01, 10, 11\}$. Thus, this set of four tests is a 2-way test set for the three parameters. If all failures of the system are triggered by faulty interactions between at most two parameters, this test set would allow all the failures to be exposed. Note that an exhaustive test set for this system would consist of $2^3 = 8$ tests.

Existing work on t-way testing has mainly focused on pairwise testing, which aims to detect faults that are caused by interactions between any two parameters. However, faults can also be caused by interactions among more than two parameters [3,4]. In order to effectively detect those faults, it is necessary to use a higher strength of coverage. This paper presents two strategies for multi-way testing (i.e. t-way testing with $t > 2$). The first strategy, called in-parameter-order-general (IPOG), is a generalization of an existing testing strategy, called in-parameter-order (IPO), from pairwise testing to multi-way testing. Strategy IPOG needs to explicitly enumerate all possible combinations of parameter values that need to be covered. (In the remainder of this paper, *a combination of parameter values* will be simply referred to as *a combination* if the context precludes any ambiguity.) While explicit enumeration is considered manageable for pairwise testing, where the number of pairwise combinations is modest even for reasonably large system configurations, it is

$$\begin{pmatrix} 0\,0\,0 \\ 0\,1\,1 \\ 1\,0\,1 \\ 1\,1\,0 \end{pmatrix}$$

Figure 1. An example of a pairwise test set.

DOI: 10.1002/stvr

an important concern for multi-way testing. This is because the number of t-way combinations (i.e. combinations involving t parameters) grows exponentially as the strength of coverage t increases. For example, consider a system with 20 parameters, where each parameter has 5 values. There are 4750 2-way combinations (or pairs), 142 500 3-way combinations, 3 028 125 4-way combinations, and 48 450 000 5-way combinations. To alleviate this problem, the second strategy, which will be referred to as IPOG-D, combines the IPOG strategy with an approach, called doubling construction (or D-construction), to reduce the number of t-way combinations that have to be enumerated. The D-construction approach is a recursive construction procedure that allows a larger test set to be constructed using smaller ones, and it does not involve any enumeration of combinations. Both strategies are deterministic, i.e. they always produce the same test set for the same system configuration. This paper reports a multi-way testing tool, called FireEye, and discusses how to efficiently implement strategies IPOG and IPOG-D. An analytical and experimental evaluation of the two strategies is also provided, including a comparison with several existing tools that support multi-way testing.

The remainder of the paper is organized as follows. Section 2 briefly reviews existing work on t-way testing. Section 3 describes the IPOG strategy and presents an IPOG-based test generation algorithm. Section 4 describes the IPOG-D strategy and presents an IPOG-D-based test generation algorithm. An analytic comparison of the IPOG-D strategy with the IPOG strategy is also provided in Section 4. Section 5 describes the FireEye tool and discusses how to efficiently implement the two strategies. Section 6 reports some experimental results. Section 7 provides concluding remarks and a plan for further work.

2. RELATED WORK

Existing approaches to t-way testing can be classified as either computational or algebraic [2,7]. Computational approaches involve explicitly enumerating all of the possible combinations, whereas algebraic approaches construct test sets based on some pre-defined rules without enumerating any combinations. Sections 2.1 and 2.2 briefly review the two types of approaches.

2.1. Computational approaches

Cohen *et al*. first proposed a strategy, called AETG (automatic efficient test generator), which builds a test set 'one-test-at-a-time' until all the combinations are covered [8,9]. A greedy algorithm is used to construct the tests such that each subsequent test covers as many uncovered combinations as possible. Several variants of this strategy have been reported that use slightly different heuristics in the greedy construction process [5,10]. The AETG strategy and its variants [5,10] are later generalized into a general framework [11]. A pairwise testing strategy, called IPO, builds a pairwise test set for the first two parameters, extends the test set to cover the first three parameters, and continues to extend the test set until it builds a pairwise test set for all the parameters [12,13]. Covering one parameter at a time allows IPO to achieve a lower order of complexity than AETG [13]. More recently, techniques such as hill climbing and simulated annealing have been applied to t-way testing [2]. These techniques start from a pre-existing test set and then apply a series of transformations to the test set until a test set is reached that covers all the combinations that

DOI: 10.1002/stvr

need to be covered. This is in contrast with AETG and IPO, which allow a test set to be built from scratch. Note that AETG and IPO can also start from a pre-existing test set, but they only extend the pre-existing test set. That is, they add new tests and/or extend existing tests with new values, without changing the values in the pre-existing test set. Techniques like hill climbing and simulated annealing can produce smaller test sets than AETG and IPO, but they typically take longer to complete. Note that the results for simulated annealing have only been reported up to 3-way testing [2].

The main advantage of computational approaches is that they can be applied to an arbitrary system configuration. That is, there is no restriction on the number of parameters and the number of values each parameter can take in a system configuration. Moreover, it is relatively easy to adapt computational approaches for test prioritization [14] and constraint handling [15]. However, computational approaches involve explicitly enumerating all possible combinations to be covered. When the number of combinations is large, explicit enumeration can be prohibitive in terms of both the space for storing these combinations and the time needed to enumerate them. In addition, computational approaches are typically greedy, i.e. they construct tests in a locally optimized manner, which does not necessarily lead to a globally optimized test set. Thus, the size of test sets generated may not be minimal.

2.2. Algebraic approaches

In algebraic approaches, test sets are derived from covering arrays that are constructed by predefined rules without requiring any explicit enumeration of combinations. There are two main types of algebraic approach for constructing covering arrays. In the first type of approach, a covering array is constructed directly by computing a mathematical function for the value of each cell based on its row and column indices. These approaches are generally extensions of the mathematical methods for constructing orthogonal arrays [16,17]. The second type of approach is based on the idea of recursive construction, which allows larger covering arrays to be constructed from smaller covering arrays [18,19]. For example, the D-construction approach uses a pair of 2-way and 3-way covering arrays of k columns to construct a 3-way covering array of $2k$ columns [20]. The details of the D-construction approach will be discussed in Section 4.

Since algebraic approaches do not enumerate any combinations, they are immune to any combinatorial effect. The computations involved in algebraic construction are usually lightweight. Thus, algebraic approaches can be extremely fast. Unfortunately, algebraic approaches often impose serious restrictions on the system configurations to which they can be applied. For example, many approaches for constructing orthogonal arrays require that the domain size be a prime number or a power of a prime number. This significantly limits the applicability of algebraic approaches for software testing. Finally, test prioritization [14] and constraint handling [15] can be more difficult for algebraic approaches.

3. THE IPOG STRATEGY

This section presents the IPOG strategy, which generalizes the IPO strategy from pairwise testing to multi-way testing. The reason why the IPO strategy is chosen is two-fold. First, as practical

DOI: 10.1002/stvr

applications may have arbitrary configurations, it is important for a strategy to place no restrictions on the system configuration. This consideration favors computational approaches over algebraic approaches. Second, due to the combinatorial effect, multi-way testing often needs to deal with a large number of combinations. Thus, multi-way testing places a stringent demand on the time and space requirements. This consideration favors the IPO strategy over other strategies such as AETG and techniques like hill climbing and simulated annealing. Note that the IPO strategy is deterministic, i.e. it always produces the same test set for the same system configuration.

The framework of the IPOG strategy can be described as follows: For a system with t or more parameters, the IPOG strategy first builds a t-way test set for the first t parameters, it then extends the test set to a t-way test set for the first $t+1$ parameters, and then continues to extend the test set until it builds a t-way test set for all the parameters. The extension of an existing t-way test set for an additional parameter consists of two steps: (a) horizontal growth, which extends each existing test by adding one value for the new parameter; and (b) vertical growth, which adds new tests, if necessary, to the test set produced by horizontal growth.

Figure 2 shows a test generation algorithm called IPOG-Test that implements this strategy. The algorithm takes two arguments: (1) an integer t specifying the strength of coverage and (2) a parameter set *ps* containing the input parameters and their values. The output of this algorithm is a t-way test set for the parameters in set *ps*. The number of parameters in set *ps* is assumed to be

```
Algorithm IPOG-Test (int t, ParameterSet ps)
{
1.  initialize test set ts to be an empty set
2.  sort the parameters in set ps in a non-increasing order of their domain sizes, and denote
    them as P1, P2, …, and Pk
3.  add into test set ts a test for each combination of values of the first t parameters
4.  for (int i = t + 1; i ≤ k;  i ++){
5.     let π be the set of all t-way combinations of values involving parameter Pi
          and any group of (t -1) parameters among the first i – 1 parameters
6.     // horizontal extension for parameter Pi
7.     for (each test τ = (v1, v2, …, vi-1) in test set ts) {
8.        choose a value vi of Pi and replace τ with τ' = (v1, v2, …, vi-1, vi) so that τ' covers the
             most number of combinations of values in π
9.        remove from π the combinations of values covered by τ'
10.    } // end for at line 7
11.    // vertical extension for parameter Pi
12.    for (each combination σ in set π){
13.       if (there exists a test τ in test set ts such that it can be changed to cover σ) {
14.          change test τ to cover σ
15.       } else {
16.          add a new test to cover σ
17.       } // end if at line 13
18.    } // end for at line 12
19. }// end for at line 4
20. return ts;
}
```

Figure 2. Algorithm IPOG-Test.

DOI: 10.1002/stvr

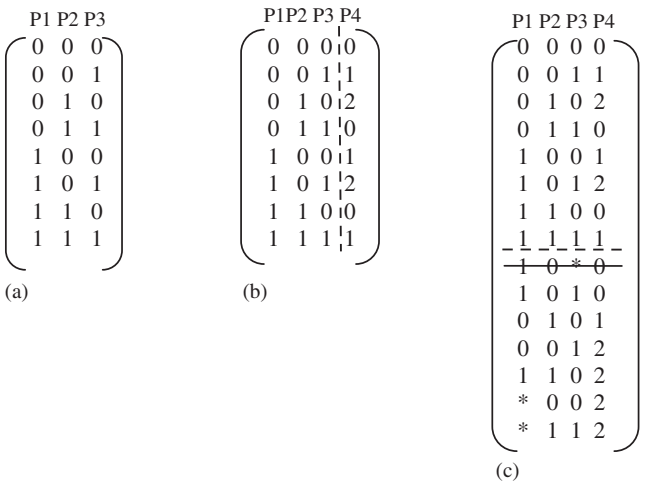

Figure 3. An illustration of algorithm IPOG-Test for a system with three Boolean parameters and one 3-valued parameter.

greater than or equal to t. Figure 3 shows an application of algorithm IPOG-Test to an example system for 3-way testing. This example system consists of four parameters, P_1, P_2, P_3, and P_4, where P_1, P_2, P_3 have two values, 0 and 1, and P_4 has three values, 0, 1, and 2. In the following, this application will be used as a running example to explain how algorithm IPOG-Test works.

Algorithm IPOG-Test begins by initializing test set *ts* to be empty (line 1) and sorting the input parameters in a non-increasing order of their domain sizes (line 2). Note that test set *ts* will be used to hold the resulting test set. Next, the algorithm builds a t-way test set for the first t parameters. This is trivially done by adding to set *ts* a test for every combination of the first t parameters (line 3). In Figure 3, the 3-way test set built for the first three parameters is shown in part (a), which contains all the eight possible combinations of the first three parameters, i.e. P_1, P_2, and P_3. Note that in Figure 3, the four parameters are not sorted for the purpose of demonstration, as otherwise vertical growth would not be needed in the test generation process.

If the number k of parameters is greater than the strength t of coverage, the remaining parameters are covered, one at each iteration, by the outermost *for* loop (line 4). Let P_i be the parameter that the current iteration is trying to cover. Covering parameter P_i means that test set *ts* is extended to become a t-way test set for parameters $\{P_1, P_2, \ldots, P_i\}$. Algorithm IPOG-Test first computes the set π of combinations that must be covered in order to cover parameter P_i (line 5). Note that test set *ts* is already a t-way test set for parameters $\{P_1, P_2, \ldots, P_{i-1}\}$. In order to build a t-way test set for $\{P_1, P_2, \ldots, P_i\}$, it is sufficient to cover all the t-way combinations involving P_i and any group of $(t-1)$ parameters among $P_1, \ldots,$ and P_{i-1}, which are the parameters that have already been covered. For example, in Figure 3, in order to cover P_4, it is sufficient to cover all the 3-way combinations of the parameter groups (P_1, P_2, P_4), (P_1, P_3, P_4), and (P_2, P_3, P_4). Each of the combinations in these parameter groups will not be listed, as these combinations can be easily enumerated. Instead, it is pointed out that each of these groups has 12 combinations. Thus, there are in total 36 combinations in set π in Figure 3.

DOI: 10.1002/stvr

The combinations in set π are covered in the following two steps:

- *Horizontal growth*: This step adds a value for parameter P_i to each of the existing tests already in test set *ts* (lines 7–10). These values are chosen in a greedy manner, i.e. at each step, the value chosen is a value that covers the largest number of combinations in set π (line 8). Note that a tie needs to be broken consistently to ensure that the resulting test set is deterministic. After a value is added, the set of combinations covered due to this addition is removed from set π (line 9). For example, in Figure 3, the fourth test is extended by adding the value 0 for P_4, which covers three combinations in set π: $\{(P_1.0, P_2.1, P_4.0), (P_1.0, P_3.1, P_4.0), (P_2.1, P_3.1, P_4.0)\}$. Here, the notation $P_i.v$ indicates that v is a value of parameter P_i. Note that if the fourth test was extended by adding the value 1 for P_4, it would only cover two combinations in set π: $\{(P_1.0, P_2.1, P_4.1), (P_2.1, P_3.1, P_4.1)\}$. The reason is that the combination $(P_1.0, P_3.1, P_4.1)$ was covered by the second test and thus was removed from set π when the second test was extended.
- *Vertical growth*: This step covers the remaining uncovered combinations, one at a time, either by changing an existing test (line 14) or by adding a new test (line 16). Before the details of this step are explained, it is necessary to introduce the notion of a *don't care* value. A *don't care* value is a value that can be replaced by any value without affecting the coverage of a test set. As shown later, *don't care* values can be introduced when a new test is added into the test set. Let σ be a t-way combination that involves parameters $P_{k_1}, P_{k_2}, \ldots,$ and P_{k_t}. An existing test τ can be changed to cover σ if and only if the value of $P_{k_i}, 1 \leq i \leq t$, in τ is either the same as in σ or a *don't care* value. In other words, when changing a test to cover a combination, only *don't care* values can be changed. If no existing test can be changed to cover σ, a new test needs to be added in which the value of $P_{k_i}, 1 \leq i \leq t$, is assigned the same value as in σ, and the other parameters are assigned *don't care* values. For example, in Figure 3, after horizontal growth, the combination $(P_1.1, P_2.0, P_4.0)$ has not been covered yet. No existing test can be found such that it can be changed to cover this combination. Thus, a new test $(P_1.1, P_2.0, P_3.*, P_4.0)$ is created to cover this combination, where '$*$' denotes a *don't care* value. This is the ninth test (i.e. the test that is crossed) in part (c). Also note that combination $(P_2.0, P_3.1, P_4.0)$ was not covered either after horizontal growth. This combination can be covered by changing the value of P_3 from '$*$' to 1 in the ninth test. In Figure 3, the ninth test is crossed out, indicating that it is replaced by the next test.

Now consider the complexity of algorithm IPOG-Test. The space complexity is dominated by the storage of π (line 5) for covering each new parameter. Let k be the number of parameters and d the largest domain size. Since the current parameter is involved in every combination, there are at most $\binom{k-1}{t-1}$ combinations of parameters in set π. Note that $\binom{k-1}{t-1} \leq (k-1) \times (k-2) \times \cdots \times (k-t-1)$, which is in $O(k^{t-1})$. Also note that each combination of parameters has at most d^t combinations of values. Thus, the space requirement for π is $O(d^t \times k^{t-1})$. Note that if all the parameters have the same domain size, the space complexity is actually dominated by the storage of π for covering the last parameter. The time complexity is dominated by horizontal extension. In Section 5, a data structure is described for storing all the combinations. With this data structure, it takes $O(1)$ time to determine whether or not a given t-way combination is already covered, and it takes $O(k^{t-1})$ time to determine the total number of t-way combinations covered where

DOI: 10.1002/stvr

a new parameter is included. Thus, it takes $O(d \times k^{t-1})$ to determine which value of the new parameter covers the most t-way combinations. As shown in [9] and supported by the experiments in Section 6, the number of tests generated by algorithm IPOG-Test is in $O(d^t \times \log k)$. Thus, the time complexity of horizontal extension, as well as that of the entire algorithm, is $O(d^{t+1} \times k^{t-1} \times \log k)$.

4. THE IPOG-D STRATEGY

The IPOG strategy involves explicitly enumerating all possible t-way combinations. Owing to the combinatorial effect, the number of combinations is often very large for multi-way testing. Thus, the cost of enumerating all possible multi-way combinations can be prohibitive in terms of both the space for storing these combinations and the time needed to enumerate them. This section presents a new strategy, called IPOG-D, for multi-way testing. This new strategy combines the IPOG strategy with a recursive construction approach, called D-construction, reducing the number of combinations that need to be enumerated. Section 4.1 describes the D-construction approach. Section 4.2 presents an IPOG-D-based test generation algorithm. Section 4.3 provides an analytic comparison between strategies IPOG and IPOG-D.

4.1. The D-construction approach

The D-construction approach is a recursive procedure that can be used to double the number of parameters in a 3-way test set [20]. It is assumed that all parameters have the same domain size, i.e. they take the same number of values. Figure 4 shows an algorithm called DoublingConstruct that implements the D-construction approach. This algorithm takes as input a 3-way test set *ts1* and a pairwise test set *ts2* for the same k parameters, and constructs as output a 3-way test set *ts* for $2k$ parameters (of the same domain size). Figure 5 shows an example that demonstrates the use of this algorithm. In Figure 5, *ts1* is a 3-way test set and *ts2* is a pairwise test set for three Boolean parameters. The resulting test set *ts* is a 3-way test set for six Boolean parameters. Again, the two values of a Boolean parameter are denoted by 0 and 1.

To understand this algorithm, it helps to view a test set as a two-dimensional array in which each row represents a test and each column represents a parameter. The terms 'row' and 'test' will be used interchangeably and likewise for terms 'column' and 'parameter'. There are two main steps in the construction process. The first step duplicates each column in *ts1* (lines 4–11). This is done by copying, row by row, the columns in *ts1* to the odd (line 7) and even (line 8) columns of the resulting test set *ts*. For example, in Figure 5, the third row (0 1 0) in test set *ts1* becomes the third row (00 11 00) in test set *ts*. Let A be a column in test set *ts1*, and B and C be the odd and even columns in test set *ts* that are copied from A, respectively. Column A will be referred to as the parent column of B and C, and columns B and C as the twin columns of each other. For example, in Figure 5, column 1 in test set *ts1* is the parent column of columns 1 and 2, in test set *ts*, where columns 1 and 2 are twin columns of each other. The values of twin columns will be written together without any space between them. Also, the resulting test set created by this step will be referred to as the *expanded* copy of test set *ts1*. In Figure 5, the expanded copy of test set *ts1* is enclosed by the parentheses in the upper part of test set *ts*.

DOI: 10.1002/stvr

```
Algorithm DoublingConstruct (TestSet ts1, TestSet ts2){
1.  let ts be an empty test set
2.  let k be the number of columns in ts1 (and ts2)
3.  // duplicate each column in ts1
4.  for (each test τ in ts1) {
5.     let τ' be a test of size 2k
6.     for (int i = 1; i <= k; i ++) {
7.        τ'[2i-1] = τ[i];
8.        τ'[2i] = τ[i];
9.     } // end for at line 6
10.    add τ' into ts;
11. } // end for at line 4
12. // add d – 1 copies of ts2
13. let d be the domain size
14. for (int i = 1; i <= d - 1; i ++) {
15.    for (each test τ in ts2) {
16.       let τ' be a test of size 2k
17.       for (int j = 1; j <=  k; j ++) {
18.          τ'[2j-1] = τ[j];
19.          τ'[2j] = (τ[j] + i) mod d;
20.       } // end for at line 17
21.       add τ' into ts
22.    } // end for at line 15
23. } // end for at line 14
24. return ts;
}
```

Figure 4. Algorithm DoublingConstruct.

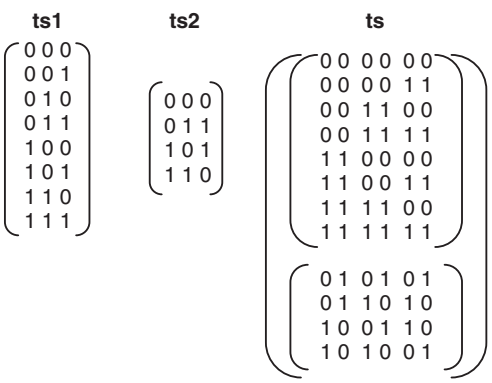

Figure 5. An illustration of algorithm *DoublingConstruct*.

Observe that the expanded copy of test set *ts1* does not cover a 3-way combination if this combination involves a pair of twin columns and the twin columns have different values. For example, let (01 0) be a 3-way combination involving the first three columns in the expanded

DOI: 10.1002/stvr

copy of test set *ts1*. This combination is not covered in the expanded copy, as the first two columns are twin columns, and the first column has value 0, while the second column has value 1. Let d be the domain size of the parameters. To cover those missing combinations, the second step adds into the resulting test set $(d-1)$ *expanded* copies of *ts2* (lines 14–23). An expanded copy of *ts2* is constructed in a way that is similar to the expanded copy of *ts1*, except that the even columns are created as a transformation, instead of an exact copy, of their parent columns (line 19). Suppose that the ith copy of *ts2* is being constructed. When an even column is copied from its parent column, each value v is mapped to $(v+i) \bmod d$. For example, in Figure 5, since all the parameters are Boolean parameters, whose domain size is 2, one expanded copy of *ts2* is added into test set *ts* and is shown in the pair of curly braces in the lower part of test set *ts*. Note that the first row (0 0 0) in *ts2* is transformed to the first row (01 01 01) in the expanded copy of *ts2*. Also note that the combination (01 0) that was not covered in the expanded copy of test set *ts1* is now covered due to this expansion.

To see why test set *ts* is a 3-way test set for $2k$ parameters, let us pick up three arbitrary columns, say, A, B, and C, from *ts* and check whether they contain all possible combinations involving the three columns. Assume that column A appears before column B and column B appears before column C. There are three cases to consider: (1) None of the three columns are twin columns of each other; (2) Columns A and B are twin columns; and (3) Columns B and C are twin columns. Note that since twin columns must be next to each, A and C cannot be twin columns. Since cases (2) and (3) are symmetric, it is sufficient to consider cases (1) and (2).

- *Case (1)*: Columns A, B, and C are copied from distinct columns in *ts1*. Since *ts1* is a 3-way test set, test set *ts* must contain all combinations involving columns A, B, and C.
  For example, in Figure 5, columns 2, 3, and 6 in test set *ts* are copied from the three different columns 1, 2, and 3 in test set *ts1*, respectively. It is easy to see that test set *ts* contains all combinations involving columns 2, 3, and 6.
- *Case (2)*: Let (a, b, c) be a 3-way combination involving columns A, B, and C. If $a = b$, the combination must be contained in the expanded copy of test set *ts1* (i.e. the upper part of test set *ts*). Otherwise, let $e = b - a$ if $b > a$, or $e = b + d - a$ if $b < a$, where d is the domain size. Then, the combination must be contained in the eth expanded copy of *ts2*. Note that e will be referred to as the difference between the values in a pair of twin columns or simply the difference between a pair of columns. For example, in Figure 5, the first two columns of test set *ts* are twin columns. Let $(a = 1, b = 1, c = 0)$ be a combination involving the first three columns. Since $a = b$, this combination is covered by the fifth row (as well as the sixth row) in the expanded copy of test set *ts1*. As another example, let $(a = 1, b = 0, c = 0)$ be a combination involving the first, second, and fifth columns. Since $b < a$, this combination is covered by the last row in the first and the only expanded copy of *ts2* (i.e. the eth expanded copy where $(e = 0 + 2 - 1 = 1)$).

Now the space and time complexity of algorithm *DoublingConstruct* is considered. Let $|ts1|$ and $|ts2|$ be the sizes of *ts1* and *ts2*, respectively, and let d be the domain size. Then, both the space and time complexities of algorithm *DoublingConstruct* are $O(|ts1| + d \times |ts2|)$. It is stressed that algorithm *DoublingConstruct* does not enumerate any combinations, and thus does not suffer from the combinatorial effect.

DOI: 10.1002/stvr

4.2. Algorithm IPOG-D-Test

Figure 6 shows a test generation algorithm called IPOG-D-Test that implements the IPOG-D strategy. Algorithm IPOG-D-Test takes the same input as algorithm IPOG-Test, and also produces a t-way test set for the parameters in set *ps*, except that t is assumed to be greater than or equal to 3. The algorithm can be divided into two parts. The first part (lines 1–4) adapts and applies algorithm *DoublingConstruct* to build an initial test set *ts*, which covers a subset of t-way combinations. The second part (lines 5–12) adapts and applies the IPOG strategy to cover the remaining t-way combinations, if they exist, making test set *ts* a complete t-way test set. Since D-construction does not enumerate any t-way combination, the IPOG-D strategy reduces the total number of combinations that need to be enumerated. In the following, the two parts are explained in detail.

Part I: The main idea of this part can be described as follows: It first builds a t-way test set *ts1* and a $(t-1)$-way test set *ts2* for half of the parameters in set *ps*. These two test sets are given to the D-construction procedure to build a test set *ts* for all the parameters. Recall that D-construction assumes that all parameters have the same domain size. One approach to handling parameters with different domain sizes is to add *don't care* values to parameters with smaller domain sizes so that all the parameters have the same domain size. In the algorithm, a finer approach is taken. Observe that since each column is duplicated separately, what is really assumed in the D-construction procedure is that each pair of twin columns must have the same domain size. To reduce the number of *don't care* values that have to be added, it is desirable to minimize the difference in domain size between each pair of twin columns. This is accomplished as follows. First, it places the parameters

Algorithm *IPOG-D-Test* (int *t*, ParameterSet *ps*)
{
1. sort the parameters in *ps* in a non-increasing order of their domain sizes, and denote them as $P_1, P_2, \ldots,$ and P_k.
2. divide the parameters into two groups: $G_1 = \{P_{2i-1} \mid 1 \le i \le \left\lceil \frac{k}{2} \right\rceil\}$ and $G_2 = \{P_{2i} \mid 1 \le i \le \left\lfloor \frac{k}{2} \right\rfloor\}$.
3. construct a *t*-way test set *ts1* and a $(t-1)$-way test set *ts2* for the parameters in *G1*.
4. let *ts = DoublingConstruct (ts1, ts2);*
5. **if** (*t* > *3*) {
6. let *covered* be a set of parameters initialized to be *G1*.
7. **for** (each parameter *P* in *G2*) {
8. let π be the set of missing tuples involving *P* and *t-1* parameters in *covered;*
9. extend test *ts* to cover the combinations in π
10. add *P* into *covered;*
11. }
12. }
13. **return** *ts*;
}

Figure 6. Algorithm IPOG-D-Test.

DOI: 10.1002/stvr

in a non-increasing order of their domain sizes (line 1). Next, it divides the parameters into two groups: G_1 and G_2 (line 2). Group G_1 contains all the parameters with odd indices, and group G_2 contains all the parameters with even indices. (The indices are assumed to start from 1.) The parameters in G_1 are used to build test sets *ts1* and *ts2*, while the parameters in G_2 are covered as a result of the doubling process. Therefore, twin columns in the resulting test set *ts* are next to each other after sorting, which minimizes the difference in domain size between a pair of twin columns.

Note that in line 3 test sets *ts1* and *ts2* can be constructed recursively using the IPOG-D strategy if the number of parameters in G_1 is large. Otherwise, they can be constructed directly using the IPOG strategy. Also note that when algorithm *DoublingConstruct* is called to double the columns in test sets *ts1* and *ts2*, three modifications need to be made. First, whenever it copies (line 8 in Figure 4) or transforms (line 19 in Figure 4) a value v to an even column, it is necessary to check whether value v is a valid value in the even column. Value v is assigned if it passes this check; otherwise, a '$*$' (i.e. *don't care*) value is assigned. Second, since there is no uniform domain size, it takes the largest domain size in line 13 of Figure 4. Note that, after sorting, the largest domain size is that of the first parameter. Finally, if the total number of parameters in set *ps* is odd, the number of parameters in G_1 will be one more than that in G_2. In this case, the last odd column simply does not need to be copied.

It is pointed out that D-construction was originally developed to construct 3-way test sets. For t-way testing where $t>3$, it may not cover all the t-way combinations. This means that test set *ts* resulting from the above procedure may not be a complete t-way test set. Test set *ts* is made complete in the second part, which is described next.

Part II: In this part, algorithm IPOG-Test is applied, with necessary adaptations, to make test set *ts* complete for t-way testing, where $t>3$ (lines 5–12). (If $t=3$, algorithm *IPOG-D-Test* is degraded to D-construction.) Set *covered* is used to represent the set of parameters that have already been covered. Since test set *ts1* is a t-way test set for the parameters in G_1, and test set *ts* is constructed by doubling the columns in *ts1*, *ts* is also a t-way test set for the parameters in G_1. Thus, set *covered* is initialized to be G_1 (line 6). Each iteration of the *for* loop covers one of the remaining parameters, i.e. those in G_2. The *for* loop will not be explained line by line, as it is largely the same as the outermost *for* loop (Figure 2, line 4) in algorithm *IPOG-Test*. Instead, the following two major differences between the two loops are pointed out:

- The computation of set π (Figure 6, line 8) needs to exclude those combinations that are already covered by the first part. The details of how to compute set π are discussed later in this section and in Section 5.1.
- In algorithm IPOG-D-Test, an existing test is extended during horizontal growth only if the current parameter being covered does not have a valid value or its value is *don't care* in the test. This is different from algorithm *IPOG-Test*, where every existing test is extended during horizontal growth. The reason for this difference is that some of the existing tests added in the first part already have a valid value for the current parameter being covered, and thus do not need to be extended.

Now, the computation of the set π of combinations (line 8, Figure 6) is discussed. Let σ be a t-way combination that involves the current parameter P being covered and a group of $(t-1)$ parameters in set *covered*. Let *columns* be the set of columns corresponding to these parameters. Consider the

DOI: 10.1002/stvr

following cases:

- *Case 0*: Set *columns* contains no twin columns;
- *Case 1*: Set *columns* contains one pair of twin columns;
- ...;
- *Case* $\lfloor t/2 \rfloor$: Set *columns* contains $\lfloor t/2 \rfloor$ pairs of twin columns.

For case (0), σ must be covered in test set *ts* by the D-construction procedure, as discussed in Section 4.1. For the remaining cases, σ is covered in test set *ts* if the difference between any pair of twin columns in σ is uniform. (Note that the difference between a pair of twin columns in a combination is defined in case 2, Section 4.1. Assume that a combination has w pairs of twin columns. Let d_i be the difference between the two values in the ith pair. These differences are uniform if $d_1 = d_2 = \cdots = d_w$.) Also note that case (1) is a special case, in that there is only one pair of twin columns in case (1). Thus, any combination in case (1) must be covered by the D-construction procedure. Therefore, set π can be computed by (a) deriving all the combinations of parameters that satisfy the conditions in cases 2 to $\lfloor t/2 \rfloor$ and (b) for each combination of parameters, deriving all possible combinations of values in which the difference between the values of any pair of twin columns is not uniform.

Note that the space and time complexity of algorithm IPOG-D-Test is dominated by Part II, and thus is the same as that of algorithm IPOG-Test.

4.3. An analytic comparison

This section sheds some light on the number of combinations of which algorithm *IPOG-D-Test* avoids explicit enumeration, compared with algorithm *IPOG-Test*. Consider a system consisting of k parameters, where each parameter has d values. Let t be the strength of coverage. To simplify the presentation, assume that k is an even number and $k \geq 2t$. The number of combinations of parameters in case (0) is

$$C_0 = \frac{\prod_{j=0}^{t-1}(k-2j)}{t!}$$

The number of combinations of parameters in case (i), where $1 \leq i \leq \lfloor t/2 \rfloor$ is

$$C_i = \binom{k}{2}{i} \times \frac{\prod_{j=0}^{t-2i-1}(k-2(i+j))}{(t-2i)!}$$

Therefore, the number of combinations of values covered by the D-construction procedure (i.e. those in which the difference between the values of any pair of twin columns is uniform) is

$$I_D = d^t \times C_0 + \sum_{i=1}^{\lfloor t/2 \rfloor} (d^{t-i+1} \times C_i)$$

DOI: 10.1002/stvr

Assume that *ts1* and *ts2* are constructed using the IPOG strategy, i.e. not using the IPOG-D strategy recursively[‡]. The number of combinations that need to be enumerated for constructing *ts1* and *ts2* is

$$I_{ts1} = d^t \times \binom{k}{t} \quad \text{and} \quad I_{ts2} = d^{t-1} \times \binom{k}{t-1}$$

Thus, the number of combinations of which algorithm IPOG-D-Test avoids explicit enumeration is $I_R = I_D - I_{ts1} - I_{ts2}$. For example, let $k = 20, t = 5, d = 5$. Then, $I_D = 25\,200\,000 + 21\,000\,000 + 450\,000$, $I_{ts1} = 787\,500$, and $I_{ts2} = 131\,250$. Thus, $I_R = 45\,731\,250$. This means that algorithm IPOG-D-Test can successfully avoid explicit enumeration of more than 45 million combinations for this system. Note that the number of all possible 5-way combinations for this system is 48 450 000. Thus, algorithm IPOG-D-Test avoids explicit enumeration of 94% of the combinations for this system.

It is stressed that the subset of t-way combinations covered by D-construction can be nicely characterized, i.e. it is known exactly what t-way combinations are guaranteed to be covered, without enumerating them. If explicit enumeration were required to find out which t-way combinations would be covered by D-construction, which is needed to determine the starting point of the subsequent application of the IPOG strategy, the IPOG-D strategy would yield no reduction on the total number of combinations to be enumerated.

5. FIREEYE: A MULTI-WAY TESTING TOOL

A multi-way testing tool, called FireEye, has been built, which implements strategies IPOG and IPOG-D. FireEye is written in Java and has the following major components: (1) *CombinatoricsHelper*, which is a utility class responsible for all the computations related to combinatorics; (2) *CombinationManager*, which manages the combinations in a way such that they can be stored and checked efficiently; (3) *TestEngine*, which implements algorithms *IPOG-Test* and *IPOG-D-Test*; and (4) *TestGenerator*, which drives the entire test generation process. FireEye can be downloaded at http://ranger.uta.edu/∼ylei/fireeye/.

Owing to the combinatorial effect, multi-way testing often needs to deal with a large number of combinations. To enable an efficient implementation of strategies IPOG and IPOG-D, these combinations must be managed carefully. Section 5.1 discusses how t-way combinations are computed in FireEye. Section 5.2 describes the data structure used by FireEye for storing these combinations.

5.1. Computing t-way combinations

First, the computation of set π in line 5 of Figure 2 is discussed. Consider the following general problem: How to compute all n-way combinations of values of m parameters, where $n \leq m$? Conceptually, this problem needs to be solved in two steps. First, it is necessary to generate all possible combinations of n parameters out of m parameters. Second, for each combination of n parameters,

[‡]Note that explicit enumeration of more combinations would be avoided if the IPOG-D strategy were used recursively.

DOI: 10.1002/stvr

it is necessary to enumerate all possible combinations of values of these n parameters. Note that a *combination of parameters* will be referred to as a *parameter combination*, and a *combination of values* as a *value combination*.

One approach to generating combinations of n elements is to use a nested loop of n levels, each iterating through the possible values of each element. This approach can be applied to generate parameter combinations, with care given to avoid generating the same combination of parameters in different orders, and to enumerate all value combinations of a given parameter combination. This approach, however, suffers from the problem that such a nested loop must be hard coded. As described below, FireEye uses a generic approach that allows parameter and value combinations to be generated without hard coding any loops.

First consider how to generate parameter combinations. Central to the generic approach is the use of parameter vectors. A parameter vector has m dimensions, one for each parameter. Consider each parameter vector to represent a parameter combination as follows: Each dimension takes on a binary value, 0 or 1, which indicates whether the corresponding parameter is excluded or included, respectively, in the parameter combination. For example, assume that there are five parameters $\{P_1, P_2, P_3, P_4, P_5\}$. Then, a parameter vector 10101 represents a parameter combination $\{P_1, P_3, P_5\}$. Thus, the problem of generating all the n-way parameter combinations out of m parameters is transformed to the problem of generating all the parameter vectors of m dimensions in which the number of 1's is exactly n.

One naïve approach to solving the above problem is to enumerate all possible parameter vectors of m dimensions, and then filter out those in which the number of 1's is not exactly n. This enumeration can be accomplished as follows. Consider each vector to represent a numeric value, where each dimension represents a digit whose base is 2 and the significance of the digits decreases from left to right. Starting from a vector of all 0's, whose numeric value is 0, all the parameter vectors can be enumerated by repeatedly adding 1 until a vector of all 1's is reached. The addition of 1 to a vector can be done by setting the least significant digit g whose value is 0 to 1 and changing all the digits that are less significant than g to 0. For example, let 10011 be a parameter vector. Observe that the third digit (from left) is the least significant digit whose value is 0. In order to add 1 to this vector, the third digit is changed from 0 to 1, and the last two digits are set to 0. Doing so results in a new vector 10100, whose numeric value is one greater than that of vector 10011.

Instead of enumerating all possible parameter vectors and then filtering out invalid ones, FireEye implements a more efficient approach that only generates valid vectors, i.e. those in which the number of 1's is exactly n. The framework of this approach is similar to that of the naïve approach, except for the following two differences. First, it starts from a parameter vector in which the least significant n digits are set to 1, instead of the vector of all 0's. For example, let $m=5$ and $n=3$. Then, it starts from 00111. Note that such a parameter vector is the smallest one, in terms of its numeric value, that consists of three 1's. Second, every time a new parameter vector is derived, it is ensured that the number of 1's in the current vector is preserved. There are two cases to consider, depending on whether the last digit in the vector is 1 or 0.

- *Case 1*: If the last digit is 1, the least significant digit g that is 0 and is followed by 1 is found. Then, g is changed from 0 to 1 and the digit following g from 1 to 0. For example, assume that the current vector is 01011. Then, the third digit (from left) is the least significant digit that is 0 and is followed by 1. Thus, the next parameter vector is generated by changing the third digit from 0 to 1 and the fourth digit from 1 to 0, which produces 01101. Note that this

DOI: 10.1002/stvr

new vector is the smallest one that is greater than the current vector, in terms of their numeric values, and that preserves the same number of 1's.

- *Case 2*: If the last digit is 0, the least significant digit g that is 0 and is followed by 1 is found, which is similar to Case 1. At the same time, the number of 1's, say c, that appear before g, is counted. Then, g is changed from 0 to 1, and the digits that are less significant than g are set to 0, except for the last $n-c-1$ digits, which are set to 1. For example, assume that the current vector is 10110. Then, the second digit (from left) is the least significant digit that is 0 and is followed by 1. Since the first digit is 1, $c=1$. Thus, the next parameter vector is generated by changing the second digit from 0 to 1, and by setting the third and fourth digits to 0, and the last digit to 1, which results in 11001. Note that this new vector is the smallest one that is greater than the current vector, in terms of their numeric values, and that preserves the same number of 1's.

Next consider how to enumerate all possible value combinations for each parameter combination. Similar to the way a parameter combination is considered, each value combination is considered to represent a numeric value, where each dimension represents a digit whose base is the same as the domain size of the corresponding parameter and the significance of the digits decreases from left to right. Note that different digits may have different bases. Starting from a value combination of all 0's, whose numeric value is 0, all the value combinations can be enumerated by repeatedly adding 1 until a value combination is reached in which the value of each digit is its base minus 1. The addition of 1 to a value combination can be accomplished by incrementing the least significant digit g whose value is less than its base minus 1 and setting all the digits that are less significant than g to 0. For example, assume that there are three parameters P_1, P_2, and P_3, each having three values. Let 112 be a value combination of the three parameters. The second digit is the least significant digit whose value is less than its base minus 1. Note that 1 can be added to this combination by incrementing the second digit and by setting the last digit to 0, which results in a new value combination 120.

Now consider how to compute set π in line 9 of Figure 6. The key challenge for this computation is to exclude value combinations that have already been covered by the D-construction procedure. Recall from Section 4.2 that each value combination σ in π satisfies two constraints: (1) the parameters involved in σ must contain at least two pairs of twin parameters and (2) the difference between the values of any pair of twin columns must be non-uniform. Note that a pair of twin parameters, which is also referred to as a twin pair, consists of two parameters whose corresponding columns in the test set are twin columns as defined in Section 4.1. The computation is also divided into two steps. That is, the possible parameter combinations are first computed, and then the possible value combinations for each of these parameter combinations. The above two constraints are enforced in the two steps, respectively. The second constraint is enforced simply by generating all possible valuation combinations of a parameter combination and then filtering out invalid ones. The following describes an efficient scheme to enforce the first constraint in the first step.

Assume that there are k parameters $P_1, P_2, \ldots,$ and P_k. To simplify the presentation, assume that k is even. These parameters are divided into two groups: $G_1=\{P_1, P_3, \ldots, P_{k-1}\}$ and $G_2=\{P_2, P_4, \ldots, P_k\}$. Note that P_{2i-1} and P_{2i}, where $1 \le i \le k/2$, are twin parameters. Assume that the current parameter being covered is P_{2j}, i.e. the jth parameter in G_2. Set π are in two subsets π_1 and π_2 such that $\pi=\pi_1 \cup \pi_2$, where π_1 contains parameter combinations involving both P_{2j-1} and P_{2j}, and π_2 contains parameter combinations involving P_{2j} but not P_{2j-1}.

DOI: 10.1002/stvr

First consider how to compute π_1. Let σ be a parameter combination in π_1. Note that σ must contain between two and $\lfloor t/2 \rfloor$ twin pairs. Thus, in addition to twin pair (P_{2j-1}, P_{2j}), σ must contain between one and $\lfloor t/2 \rfloor - 1$ additional twin pairs. There are in total $j-1$ possible twin pairs among those parameters that are already covered: $\{(P_1, P_2), (P_3, P_4), \ldots, (P_{2j-3}, P_{2j-2})\}$. Using the procedure described earlier, it first generates all combinations of between one and $\lfloor t/2 \rfloor - 1$ twin pairs out of these $j-1$ twin pairs. Each combination τ of twin pairs can be used to generate a subset of t-way parameter combinations in π_1 as follows. Assume that τ contains i twin pairs, where $1 \leq i \leq \lfloor t/2 \rfloor - 1$. First, it adds to τ twin pair (P_{2j-1}, P_{2j}). Now τ contains $2i+2$ parameters. If $t = 2i+2$, τ is returned as the only t-way parameter combination generated from τ. Otherwise, it generates a set of t-way parameter combinations, each of which adds $t-2i-2$ parameters to τ. The additional parameters must not contain any twin pair, and are chosen from the parameters that are already covered but have not appeared in τ. These additional parameters will be referred to as 'single' parameters. Note that the parameters that are already covered include all the parameters in the first group and the first $j-1$ parameters in the second group.

For example, let $G_1 = \{P_1, P_3, P_5, P_7\}$ and $G_2 = \{P_2, P_4, P_6, P_8\}$. Let $t = 5$. Assume that P_6 is the current parameter being covered, which is the third parameter in G_2. There are in total two twin pairs, i.e. $\{(P_1, P_2), (P_3, P_4)\}$, among the parameters that are already covered, i.e. $\{P_1, P_2, P_3, P_5, P_7, P_2, P_4\}$. Note that $\lfloor t/2 \rfloor - 1 = 1$. Thus, it generates two combinations of twin pairs, i.e. $\tau_1 = (P_1, P_2)$, and $\tau_2 = (P_3, P_4)$, each of which contains only a single twin pair. For $\tau_1 = (P_1, P_2)$, it first adds to τ_1 twin pair (P_5, P_6), which results in $\tau_1 = (P_1, P_2, P_5, P_6)$. There are three parameters, namely, $\{P_3, P_4, P_7\}$, that are already covered but have not appeared in τ_1. Each of these three parameters can be added to τ_1 to generate a 5-way combination. Thus, it generates three 5-way parameter combinations from τ_1, i.e. $\{(P_1, P_2, P_3, P_5, P_6), (P_1, P_2, P_4, P_5, P_6), (P_1, P_2, P_5, P_6, P_7)\}$. Similarly, for $\tau_2 = (P_3, P_4)$, it generates three 5-way parameter combinations, i.e. $\{(P_1, P_3, P_4, P_5, P_6), (P_2, P_3, P_4, P_5, P_6), (P_3, P_4, P_5, P_6, P_7)\}$.

The computation of π_2 is similar to that of π_1, except for the following two differences. First, when it generates combinations of twin pairs, it generates all of these combinations that contain between two and $\lfloor t/2 \rfloor$ twin pairs, instead of between one and $\lfloor t/2 \rfloor - 1$ pairs. Second, when it uses a combination τ of twin pairs to generate t-way parameter combinations, it first adds P_{2j} to τ, and then $t-2i-1$ 'single' parameters, instead of $t-2i-2$ 'single' parameters. Note that the reason for both differences is that P_{2j-1} is not involved in any parameter combination of π_2. That is, the current parameter P_{2j} is included as a single parameter. Again, consider the above example. As shown earlier, there are in total two twin pairs, i.e. $\{(P_1, P_2), (P_3, P_4)\}$, among the parameters that are already covered. Note that $\lfloor t/2 \rfloor = 2$. Thus, the only combination τ of twin pairs to be generated contains both of these pairs. Then, it adds to τ the current parameter P_6, which results in the only 5-way combination $(P_1, P_2, P_3, P_4, P_6)$ in π_2.

5.2. Storing t-way combinations

This section describes the data structure used by FireEye for storing t-way combinations. On the one hand, the storage needs to be as compact as possible. On the other hand, it should be able to quickly check whether or not a given combination has already been covered. Note that such a check is the most frequently performed operation in algorithms IPOG-Test and IPOG-D-Test.

DOI: 10.1002/stvr

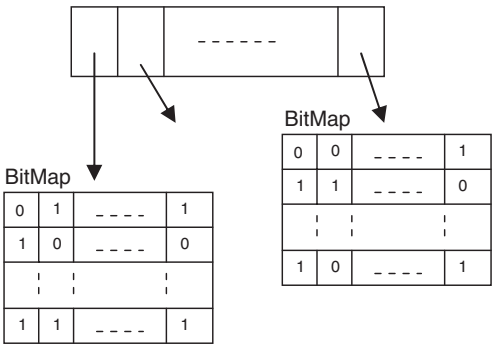

Figure 7. A two-level hierarchy for storing combinations.

As shown in Figure 7, the data structure is a hierarchy of two levels. At the first level is an array of pointers, each of which represents one possible parameter combination and points to a bitmap at the second level. Along with each pointer, the indices of the parameters involved in the corresponding combination are saved. At the second level, each bitmap has one bit for each value combination. The bit value 0 indicates that the corresponding value combination has not been covered yet, and the value 1 indicates that the corresponding value combination has already been covered. Each value combination is considered to represent a numeric value. The numeric value of a value combination is used to index the bit that corresponds to the combination.

Observe that in both IPOG-Test and IPOG-D-Test, it needs to check what value combinations are covered by an existing test if the test is extended by a given value. To do this, the array at the first level can be iterated through. For each parameter combination, the corresponding value combination, i.e. the value of each parameter in the test being extended, is first found. Next it checks in the corresponding bit map whether or not the value combination is covered. Note that since the index of a value combination in the bit map can be directly computed, this check takes $O(1)$ time.

It is worth noting that the storage hierarchy can be further optimized for algorithm IPOG-Test. The pointers can be indexed in such a way that, for a given parameter combination, it can directly compute its index and thus locate the corresponding pointer quickly without having to save the parameter combination. Consider the following example. Assume that there are four parameters, P_1, P_2, P_3, and P_4. There are four combinations of three parameters out of these four parameters, which are indexed in the following order: (P_1, P_2, P_3), (P_1, P_2, P_4), (P_1, P_3, P_4), and (P_2, P_3, P_4). The index of a given parameter combination (P_i, P_j, P_k) can be computed using the formula $3\times(i-1)+2\times(j-i-1)+(k-j-1)$. For instance, the index of (P_1, P_3, P_4) is $3\times0+2\times(3-1-1)+(4-3-1)=2$. Here, it is assumed that the index starts from 0. This indexing scheme can be easily generalized to any number of parameters.

6. EXPERIMENTAL RESULTS

The experiments can be divided into two parts. The first part is aimed to compare algorithms *IPOG-Test* and *IPOG-D-Test* with each other. In particular, it studies the growth in the size of the test

DOI: 10.1002/stvr

sets generated by algorithms *IPOG-Test* and *IPOG-D-Test* in terms of the strength of coverage, the number of parameters, and the domain size. An initial study of the effects of variations in domain sizes is also included. The second part is aimed to compare FireEye with existing tools that support multi-way testing, in terms of both the size of the resulting test set and the time needed to produce the test set. All the experiments are conducted using a laptop running Windows XP with 1.6 GHz central processing unit and having 1 GB memory.

The first part first applied FireEye to three series of system configurations. In the first series, the number of parameters is fixed to be 15, the domain size of each parameter is fixed to be 4, and the strength of coverage is varied from 3 to 6. In the second series, the strength of coverage is fixed to be 5, the domain size of each parameter is fixed to be 4, and the number of parameters is varied from 11 to 20. In the third series, the strength of coverage is fixed to be 5, the number of parameters is fixed to be 15, and the domain size is varied from 2 to 7. Note that, in order to ensure a fair comparison in terms of execution time, data structures, such as the one used for managing the combinations, are shared as much as possible between the implementations of two algorithms.

Tables I, II, and III show the experimental results for the three series of system configurations, respectively. Column 'Size Ratio' contains the ratios of the sizes of test sets generated by algorithm IPOG-D-Test over the sizes of test sets generated by algorithm IPOG-Test; column 'Time Ratio' contains the ratios of the execution times taken by algorithm IPOG-D-Test over those taken by

Table I. Results for 15 4-value parameters for 3-, 4-, 5-, and 6-way testing.

	IPOG-Test		IPOG-D-Test			
t-Way	Size	Time	Size	Size ratio	Time	Time ratio
3	181	0.56	207	1.14	0.17	0.3
4	924	16.57	1165	1.26	0.8	0.05
5	4519	230.2	6793	1.50	12.31	0.05
6	20384	2152.1	32724	1.61	334.36	0.16

Table II. Results for 11–20 4-value parameters for 5-way testing.

	IPOG-Test		IPOG-D-Test			
Number of parameters	Size	Time	Size	Size ratio	Time	Time ratio
11	3287	23.3	4761	1.45	2.81	0.12
12	3703	44.05	5049	1.36	4.75	0.11
13	4001	79.81	6132	1.53	6.49	0.08
14	4260	138.63	6356	1.49	9.81	0.07
15	4519	230.2	6793	1.5	12.31	0.05
16	4787	367.98	7011	1.48	17.39	0.05
17	5018	564.98	7613	1.52	22.5	0.04
18	5245	838.52	7815	1.49	29.74	0.04
19	5471	1205.69	8454	1.55	40.38	0.03
20	5685	1739.14	8606	1.51	50.70	0.03

DOI: 10.1002/stvr

Table III. Results for 15 parameters with 2–7 values for 5-way testing.

	IPOG-Test		IPOG-D-Test			
Number of values	Size	Time	Size	Size ratio	Time	Time ratio
2	134	4.08	188	1.4	0.4	0.09
3	1123	48.08	1552	1.38	2.39	0.05
4	4531	234.08	6739	1.49	12.31	0.05
5	15095	996.89	22197	1.47	55.06	0.06
6	37748	3273.42	56619	1.49	203.13	0.06
7	81814	9040.25	124186	1.52	694.74	0.08

algorithm IPOG-Test. The other columns in the three tables are self-explanatory. Note that the execution times are shown in seconds. The following observations are made about the results in these tables:

- It was shown that the size of a test set grows in $O(d^t \log k)$, where t is the strength of coverage, d is the domain size, and k is the number of parameters [9]. Curve fitting analysis was performed on the sizes of the test sets in the three tables. The analysis showed that the experimental results were consistent with the theoretical results. Note that the number of tests in a t-way test set grows very quickly as the strength of coverage t increases.
- Both size and time ratios between algorithms IPOG-D-Test and IPOG-Test increase slightly as the strength of coverage increases, as shown in Table I. Here, the time ratio for 3-way testing in Table I is considered to be an anomaly, as it is so small and is thus likely distorted by factors such as the startup time. The size ratio fluctuates slightly in the range of 1.36–1.55, when the number of parameters (in Table II) or values (in Table III) is increased. However, the time ratio seems to consistently decrease when the number of parameters is increased, and to consistently increase when the number of values is increased. Again, the time ratio in the first row of Table II is considered to be an anomaly. Note that the higher the size ratio, the more the tests that algorithm IPOG-D-Test generates, and the lower the time ratio, the faster algorithm IPOG-D-Test is, in comparison with algorithm IPOG-Test.
- When the number of parameters is large, algorithm *IPOG-D-Test* can be significantly faster than algorithm *IPOG-Test* at the cost of a modest increase in the size of the resulting test sets. For example, for 5-way testing for 20 4-value parameters (last row in Table II), algorithm *IPOG-D-Test* takes about 3% of the execution time of algorithm *IPOG-Test*, at the cost of generating about 51% more tests. This can be explained by the fact that the more the parameters, the greater the combinatorial effect is, and thus the greater the benefit that the D-construction approach brings to algorithm IPOG-D-Test. Note that since every extra test adds to the cost of test execution, a decision needs to be made about whether to spend more time on test generation or on test execution. This decision can be affected by factors such as the degree of test automation, the execution time of each test, the availability of resources at the two stages, and so on.

To study the effects of variations in domain sizes, FireEye was applied to five system configurations with mixed domain sizes. These configurations were used to conduct experiments in a previous study [11]. Table IV shows the results for the five configurations. Column 'Configuration' shows

DOI: 10.1002/stvr

Table IV. Results for five system configurations with mixed domain sizes.

		IPOG-Test		IPOG-D-Test			
Configuration	Variation	Size	Time	Size	Size ratio	Time	Time ratio
$3^4 4^5$	0.53	432	0.31	704	1.63	0.08	0.25
$5^1 3^8 2^2$	0.77	297	0.44	527	1.77	0.08	0.18
$8^2 7^2 6^2 5^2$	1.20	4254	2.98	7571	1.78	0.70	0.23
$6^6 5^5 3^4$	1.25	3034	26.14	4590	1.51	1.16	0.04
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$	2.74	5439	3.36	11 563	2.13	1.27	0.38

the parameters and values of each configuration in the following format: $d_1^{k_1} d_2^{k_2} \ldots$, indicating that there are k_1 parameters with d_1 values, k_2 parameters with d_2 values, and so on. For example, $5^1 3^8 2^2$ in the last row indicates that there is one parameter with five values, eight parameters with three values, and two parameters with two values. Column 'Variation' shows the standard deviation of the domain sizes in each configuration. The other columns are the same as those in Tables I–III. Observe that both size and time ratios are generally higher than those in Tables I–III. This suggests that variations in domain sizes may have a negative effect on the performance of algorithm *IPOG-D-Test*. A more thorough study of the effects of variations in domain sizes will be reported in a separate paper.

In the second part, two existing tools are identified that support t-way testing and are either open source or free for academic use[§]: (1) Intelligent Test Case Handler (or ITCH), which is from IBM Research [23]; and (2) Jenny, which is from www.burtleburtle.net [24]. ITCH implements a combination of several algebraic methods, and Jenny implements a greedy algorithm. The algorithmic details of ITCH and Jenny are not known. Note that ITCH is written in Java, while Jenny is written in C.

The three tools, namely, FireEye, ITCH, and Jenny, were first applied to a Traffic Collision Avoidance System (TCAS) module. The TCAS module implements part of an aircraft collision avoidance system specified by the Federal Aviation Administration, and has been used in other studies of software testing [25,26]. The module has 12 parameters: seven parameters have two values, two parameters have three values, one parameter has four values, and two parameters have ten values. Table V shows the sizes of the test sets generated by each tool and the times taken to generate these test sets from 3-way testing up to 6-way testing. The execution times are shown in seconds. Note that the results for 5- and 6-way testing for ITCH are not available (NA). This is because ITCH only supports up to 4-way testing. Also note that ITCH supports two test generation algorithms, namely, CTS and Tofu. The CTS algorithm, which is the default option, was used to collect the results; the Tofu algorithm takes much longer than the CTS algorithm. It is interesting to note that ITCH generates more tests for 3-way testing than for 4-way testing. For all the cases in Table V, algorithm IPOG-Test generated fewer tests faster than ITCH and Jenny, and algorithm

[§] Two other tools, namely, TConfig [21] and TVG [22], were also identified that support t-way testing and are free for academic use. However, their performance is substantially worse than that of the other tools, and is thus not reported here.

DOI: 10.1002/stvr

Table V. Results of FireEye, ITCH, and Jenny for the TCAS configuration.

	IPOG-Test		IPOG-D-Test		ITCH		Jenny	
t-Way	Size	Time	Size	Time	Size	Time	Size	Time
3	400	0.55	480	0.13	2388	1020	413	0.71
4	1361	4.22	2522	0.34	1484	5400	1536	3.54
5	4220	25.39	5306	4.25	NA	NA	4580	43.54
6	10 918	98.73	14 480	47.72	NA	NA	11 625	470

Table VI. Results of FireEye and Jenny for 4-way testing of several configurations.

	IPOG-Test		IPOG-D-Test		Jenny	
Configurations	Size	Time	Size	Time	Size	Time
$3^4 4^5$	432	0.31	704	0.08	457	1.05
$5^1 3^8 2^2$	297	0.44	527	0.08	303	0.80
$8^2 7^2 6^2 5^2$	4254	2.98	7571	0.70	4580	40.31
$6^6 5^5 3^4$	3034	26.14	4590	1.16	3033	162.78
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1$	5439	3.36	11 563	1.27	6198	43.55

IPOG-D-Test generated more tests, but was even faster. Note that since FireEye is written in Java, and Jenny is written in C, it is hard to directly compare their execution times.

Table VI reports some additional results of applying FireEye and Jenny to the system configurations shown in Table IV. Note that the results for FireEye were copied from Table IV. Also note that the results for ITCH are not included because ITCH did not complete after 1 hour even for configuration $3^4 4^5$. For all the cases, algorithm IPOG-Test generated fewer tests faster than Jenny, except for configuration $6^6 5^5 3^4$, where algorithm IPOG-Test generated one more test but was substantially faster than Jenny. As expected, algorithm IPOG-D-Test generated more tests, but was much faster than both algorithm IPOG-Test and Jenny.

7. CONCLUSION AND FUTURE WORK

This paper presents two test generation strategies, namely, IPOG and IPOG-D, for multi-way testing. Strategy IPOG is a generalization of an existing pairwise testing strategy to multi-way testing. This strategy explicitly enumerates all possible combinations, which may not be practical when the number of combinations is large and when resources are constrained. To address this problem, strategy IPOG-D combines IPOG with a recursive construction procedure, namely, D-construction, to reduce the number of combinations that have to be enumerated. An important view has been taken that the minimization of the resulting test sets must be balanced with time and space requirements. This is different from existing work that has largely focused on pairwise testing. For pairwise testing, the number of combinations is modest even for reasonably large system configurations. Thus, existing work is mainly concerned with minimizing the size of the resulting test sets, while paying little attention on the time and space requirements.

DOI: 10.1002/stvr

This paper considers t-way testing to be a very promising technique for generating test data for several reasons. First, as a specification-based technique, it requires no knowledge about the implementation under test. Moreover, the specification required by t-way testing can be very lightweight, as a basic system configuration only needs to identify the input parameters and the possible values of each of those parameters. Second, t-way testing has the potential to be very effective for various types of applications. Kuhn *et al*. studied actual faults in several industrial applications, showing that all of the known faults in these applications are caused by up to 6-way interactions [4]. This means that if 6-way testing is considered to those applications, all of those faults will be exposed (assuming that category partitioning or other abstraction methods are applied judiciously to identify the parameter values). Finally, the test data generation process for multi-way testing can be fully automated as a push-button feature, which is a key to industrial acceptance.

There are a number of venues for future work. First, a thorough comparison with existing results in the literature will be conducted. In particular, Colbourn maintains a catalog of best-known covering arrays [27]. It is interesting to compare the results of FireEye with Colbourn's catalog. Second, algorithms *IPOG-Test* and *IPOG-D-Test* will be extended to support parameter relations and constraints. Parameter relations are used to avoid exercising combinations between parameters that do not interact with each other. Parameter constraints are used to exclude combinations that are not permitted from the domain semantics [14,15]. Third, t-way testing often generates a large number of tests, which makes it impractical to manually execute the tests and evaluate their results. An integration of FireEye with other tools will be performed to automate the entire testing process, including test generation, test execution, and test evaluation. Finally, empirical studies will be conducted to evaluate the fault detection effectiveness of t-way testing, especially on industrial applications such as the NASA application used in the experiments in Section 6. An integrated solution that automates the entire testing process will make such empirical studies possible.

ACKNOWLEDGEMENTS

The authors would like to thank Renee Bryce and Richard Carver for their comments on an earlier version of this paper, and Chinmay Jayaswal for conducting some of the experiments reported in this paper. The authors would also like to thank anonymous reviewers for their valuable comments. This work is partly supported by a grant (Award No. 60NANB6D6192) from the Information Technology Lab (ITL) of the National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

REFERENCES

1. Grindal M, Offutt J, Andler SF. Combination testing strategies—A survey. *Journal of Software Testing, Verification and Reliability* 2004; **5**(3):167–199.
2. Cohen MB, Colbourn CJ, Gibbons PB, Mugridge WB. Constructing test suites for interaction testing. *Proceedings of 25th IEEE International Conference on Software Engineering*, Portland, Oregon, 2003; 38–48.
3. Kuhn DR, Reilly MJ. An investigation of the applicability of design of experiments to software testing. *Proceedings of 27th NASA/IEEE Software Engineering Workshop*, Greenbelt, Maryland, 2002; 91–95.
4. Kuhn DR, Wallace D, Gallo A. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 2004; **30**(6):418–421.

DOI: 10.1002/stvr

5. Tung YW, Aldiwan WS. Automating test case generation for the new generation mission software system. *Proceedings of IEEE Aerospace Conference*, Big Sky, Montana, 2000; 431–437.
6. Wallace DR, Kuhn DR. Failure modes in medical device software: An analysis of 15 years of recall data. *International Journal of Reliability*, *Quality and Safety Engineering* 2001; **8**(4):351–371.
7. Hartman A, Raskin L. Problems and algorithms for covering arrays. *Discrete Mathematics* 2004; **284**(1–3):149–156.
8. Cohen DM, Dalal SR, Parelius J, Patton GC. The combinatorial design approach to automatic test generation. *IEEE Software* 1996; **13**(5):83–87.
9. Cohen DM, Dalal SR, Fredman ML, Patton GC. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 1997; **23**(7):437–444.
10. Bryce RC, Colbourn CJ. The density algorithm for pairwise interaction coverage. *Journal of Software Testing*, *Verification and Reliability* 2007; **17**(3):159–182.
11. Bryce RC, Colbourn CJ, Cohen MB. A framework of greedy methods for constructing interaction test suites. *Proceedings of 27th IEEE International Conference on Software Engineering*, Missouri, 2005; 146–155.
12. Lei Y, Tai KC. In-parameter-order: A test generation strategy for pairwise testing. *Proceedings of 3rd IEEE International Conference on High-Assurance Systems Engineering Symposium*, Washington, DC, 1998; 254–261.
13. Tai KC, Lei Y. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering* 2002; **28**(1):109–111.
14. Bryce RC, Colbourn CJ. Prioritized interaction testing for pairwise coverage with seeding and avoids. *Information and Software Technology Journal* 2006; **48**(10):960–970.
15. Grindal M, Offutt J, Mellin J. Conflict management when using combination strategies for software testing. *Proceedings of 18th Australian Software Engineering Conference*, Melbourne, Australia, 2007.
16. Bush KA. Orthogonal arrays of index unity. *Annals of Mathematical Statistics* 1952; **23**:426–434.
17. Mandl R. Orthogonal Latin squares: An application of experiment design to compiler testing. *Communications of the ACM* 1985; **28**(10):1054–1058.
18. Williams AW. Determination of test configurations for pair-wise interaction coverage. *Proceedings of 13th International Conference on the Testing of Communicating Systems*, Ottawa, Canada, 2000; 59–74.
19. Williams AW, Probert RL. A practical strategy for testing pair-wise coverage of network interfaces. *Proceedings of 7th International Symposium on Software Reliability Engineering*, Niagara, Canada, 1996; 246–254.
20. Chateauneuf MA, Colbourn CJ, Kreher DL. Covering arrays of strength 3. *Designs*, *Codes*, *and Cryptography* 1999; **16**:235–242.
21. Williams A. TConfig. http://www.site.uottawa.ca/∼awilliam/ [21 September 2007].
22. Arshem J. TVG. http://sourceforge.net/projects/tvg/ [21 September 2007].
23. Hartman A, Klinger T, Raskin L. IBM intelligent test case handler. http://www.alphaworks.ibm.com/tech/whitch [21 September 2007].
24. Jenkins B. Jenny. http://www.burtleburtle.net/bob/math/jenny.html [21 September 2007].
25. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of 16th IEEE International Conference on Software Engineering*, 1994; 191–200.
26. Kuhn DR, Okun V. Pseudo-exhaustive testing for software. *Proceedings of 30th NASA/IEEE Software Engineering Workshop*, 2006; 153–158.
27. Colbourn CJ. http://www.public.asu.edu/∼ccolbou/src/tabby/catable.html [27 September 2007].

DOI: 10.1002/stvr