IPOG: A General Strategy for T-Way Software Testing

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

[1]Dept. of Comp. Sci. and Eng.
University of Texas at Arlington
Arlington, TX 76019
Email: ylei@cse.uta.edu

[2]Information Technology Laboratory
National Inst. of Standards and Tech.
Gathersburg, MD 20899
Email: {raghu.kacker, kuhn, vadim.okun}@nist.gov

[3]Dept. of Mathematics
George Mason University
Fairfax, VA 22030
Email: lawrence@gmu.edu

Abstract

Most existing work on *t*-way testing has focused on 2-way (or pairwise) testing, which aims to detect faults caused by interactions between any two parameters. However, faults can also be caused by interactions involving more than two parameters. In this paper, we generalize an existing strategy, called In-Parameter-Order (IPO), from pairwise testing to *t*-way testing. A major challenge of our generalization effort is dealing with the combinatorial growth in the number of combinations of parameter values. We describe a *t*-way testing tool, called FireEye, and discuss design decisions that are made to enable an efficient implementation of the generalized IPO strategy. We also report several experiments that are designed to evaluate the effectiveness of FireEye.

1. Introduction

One approach to software testing is combinatorial testing, which creates test suites by selecting values for input parameters and by combining these parameter values. For a system with *n* parameters, each of which has *d* values, the number of all possible combinations of values of these parameters is d^n. Due to resource constraints, it is nearly always impossible to exhaustively test all of these combinations of parameter values. Thus, a strategy is needed to select a subset of these combinations. One such strategy, called *t*-way testing, requires every combination of any *t* parameter values to be covered by at least one test, where *t* is referred to as the strength of coverage and usually takes a small value. 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 [6]. We can consider each combination of parameter values 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.

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 as 0 and 1. Fig. 1 shows a pairwise test set for this system. In the 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). It can be checked 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}. 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 detected. Note that an exhaustive test set for this system would consist of $2^3 = 8$ tests.

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

Figure 1. A 2-way test set for 3 boolean parameters

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 involving more than two parameters [10][11]. In order to effectively detect those faults, it is necessary to enable a higher strength of coverage. In this paper, we generalize an existing strategy, called In-Parameter-Order (or IPO), from pairwise testing to general *t*-way testing. The resulting strategy is referred to as In-Parameter-Order-General (or IPOG). A major challenge of our generalization effort is dealing with the combinatorial growth in the number of combinations of parameter-values. We describe a *t*-way testing tool called FireEye, and discuss design decisions that are made to enable an efficient implementation of the IPOG strategy. We also report several experiments that were conducted to evaluate the effectiveness of FireEye. In particular, we conducted an experiment that compared FireEye to several existing tools. The result of this experiment indicates that FireEye performed significantly better than the other tools for a real-life application.

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. An algorithm that implements the IPOG strategy is also presented in Section 3. Section 4 describes the FireEye tool, and discusses several key design decisions. Section 5 reports the design and the results of the experiments. Section 6 provides concluding remarks and our plan for further work

2. Related Work

Cohen et al. proposed a strategy, called Automatic Efficient Test Generator (or AETG), which constructs a test set by repeatedly adding one test at a time until all the combinations of parameter values are covered [4][5]. A greedy algorithm is used to construct the tests such that each test covers as many uncovered combinations as possible. Several variants of this strategy have been reported in the literature [2][17]. These variants share the same framework as AETG but use different heuristics for the greedy construction of each test [6]. In [13][16], we proposed the IPO strategy, which 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. Covering one parameter at a time allows the IPO strategy to achieve a lower order of complexity than AETG. Most recently, heuristic search techniques such as hill climbing and simulated annealing have been applied to multi-way testing [6]. Unlike AETG and IPO, which builds a test set from scratch, heuristic search 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. Heuristic search techniques can produce smaller test sets than AETG and IPO, but they typically take longer to complete.

In addition to computational approaches, algebraic approaches have also been reported. These approaches construct test sets using pre-defined rules. Some algebraic approaches compute test sets directly by a mathematical function. These approaches are generally extensions of the mathematical methods for constructing orthogonal arrays [1][14]. Informally, an orthogonal array of strength t requires that every possible combination of any t columns be covered exactly once. Therefore, an orthogonal array can be considered as an optimal t-way test set if we consider each row to represent a test and each column to represent a parameter. Other algebraic approaches are based on the idea of recursive construction, which allows larger test sets to be constructed from smaller ones [8][18].

Computational and algebraic approaches have their own advantages and disadvantages. Computational approaches can be applied to arbitrary system configurations, but they can be expensive as they involve explicit enumeration and there can be a large number of combinations to be enumerated. The computations involved in algebraic approaches are typically lightweight, and in some cases, algebraic approaches can produce optimal test sets. However, algebraic approaches often impose restrictions on the system configurations to which they can be applied.

Finally, many empirical studies have been reported on assessing the fault detection effectiveness of t-way testing. In [3], Burr and Young showed that pairwise testing achieves higher block and decision coverage than traditional methods for a commercial email system. In [7], Dalal et al. applied t-way testing to a telephone software system and showed that several faults can only be detected under certain combinations of input parameters. In [10][11], Kuhn et al. studied the actual faults in several software projects, and found that all the known faults are caused by interactions among 6 or fewer parameters.

3. The IPOG Strategy

In this section, we present the IPOG strategy. Our motivation for generalizing the IPO strategy is two-fold. First, we want to develop a testing strategy that can be applied to general software applications. Thus, the strategy should put no restrictions on the system configuration under test. This consideration favors computational approaches over algebraic approaches. (Recall from Section 2 that the former can be applied to an arbitrary system configuration, while the latter often has restrictions on the system configurations to which they can be applied.) Second, general t-way testing has a more stringent demand on the time and space requirements than pairwise testing. This is because the number of combinations grows exponentially as the strength of coverage increases. This consideration favors the IPO strategy over other strategies such as AETG and heuristic search techniques. We also 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 builds a t-way test set for the first t parameters, extends the test set to build 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 parameters can be in an arbitrary order.) The extension of an existing t-way test set for an additional parameter is done in two steps:

- *horizontal growth*, which extends each existing test by adding one value for the new parameter;
- *vertical growth*, which adds new tests, if needed, to the test set produced by horizontal growth.

Fig. 2 shows a test generation algorithm called IPOG-Test that implements this framework. The algorithm takes as input an integer t and a set *ps* of parameters, and produces as output a t-way test set for the parameters in set *ps*. It is assumed that the number n of parameters in set *ps* is greater than or equal to t. Fig. 3 shows an application of algorithm IPOG-Test to an example system for 3-way testing. This

```
Algorithm IPOG-Test (int t, ParameterSet ps)
{
1.  initialize test set ts to be an empty set
2.  denote the parameters in ps, in an arbitrary order, as $P_1, P_2, \ldots,$ and $P_n$
3.  add into test set ts a test for each combination of values of the first t parameters
4.  for (int $i = t + 1$; $i \le n$;  $i$ ++){
5.     let $\pi$ be the set of t-way combinations of values involving parameter $P_i$
          and $t$ -1 parameters among the first $i - 1$ parameters
6.     // horizontal extension for parameter $P_i$
7.     for (each test $\tau = (v_1, v_2, \ldots, v_{i-1})$ in test set ts) {
8.        choose a value $v_i$ of $P_i$ and replace $\tau$ with $\tau' = (v_1, v_2, \ldots, v_{i-1}, v_i)$ so that $\tau'$ covers the
              most number of combinations of values in $\pi$
9.        remove from $\pi$ the combinations of values covered by $\tau'$
10.    }
11.    // vertical extension for parameter $P_i$
12.    for (each combination $\sigma$ in set $\pi$){
13.       if (there exists a test that already covers $\sigma$) {
14.          remove $\sigma$ from $\pi$
15.       } else {
16.          change an existing test, if possible, or otherwise add a new test
                to cover $\sigma$ and remove it from $\pi$
17.       }
18.    }
19.}
20.return ts;
}
```

Figure 2: Algorithm IPOG-Test

example system consists of four parameters P1, P2, P3, and P4, where P1, P2, P3 have two values 0 and 1, and P4 has three values 0, 1, and 2. In the following, we will use this application 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 by putting the input parameters into an arbitrary order (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 into test set ts a test for every combination of the first t parameters (line 3). In Fig. 3, the 3-way test set built for the first three parameters is shown in part (a), which contains all the 8 possible combinations of the first three parameters, i.e., P1, P2, and P3.

If the number *n* 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. We first compute the set π of combinations that must be covered in order to cover parameter P_i (line 5). Covering parameter P*i* means extending test set *ts* so that it becomes a *t*-way test set for parameters P_1, …, P_{i-1}, and P_i. Note that test set *ts* is already a *t*-way test set for parameters P_1, …, P_{i-1}. Thus, we only need to cover all the *t*-way combinations involving P_i and any group of *t* -1 parameters among P_1, …, P_{i-1}, which are the parameters that are already covered. For example, in Fig. 3, in order to cover P4, we need to cover all the 3-way combinations of the following parameter groups, (P1, P2, P4), (P1, P3, P4), and (P2, P3, P4). We will not list each of the combination in those parameter groups, as they can easily be enumerated. Instead, we only point out that each of these groups has 12 combinations. Thus, there are in total 36 combinations in the set π computed for Fig. 3.

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

- Horizontal growth: This step extends each of the existing tests by adding a value for parameter Pi (lines 7 - 10). These values are chosen in a greedy manner, i.e., at each step, the value chosen is one that covers the largest number of combinations in set π (line 8). Each time a value is added, the set of combinations covered due to this addition are removed from set π (line 9). For example, in Fig. 3, the 4th test is extended by adding the value 0 for P4, which covers three combinations in set π: {(P1.0, P2.1, P4.0), (P1.0, P3.1, P4.0), (P2.1, P3.1, P4.0)}. Here notation Pi.v indicates that v is a value of parameter Pi. Note that if the 4th test was extended by adding the value 1 for P4, it would only cover two combinations in set π: {(P1.0, P2.1, P4.1), (P2.1, P3.1, P4.1)}. The reason is that the combination (P1.0, P3.1, P4.1) was covered by the 2nd test and thus was removed from set π when the 2nd test was extended.

- Vertical growth: This step covers the remaining uncovered combinations, one at a time, either by changing an existing test or by adding a new test (line 16). When we change a test to cover a combination, only *don't care* values can be changed. A *don't care* value is a value that can be replaced by any value without affecting the coverage of a test set. If no existing test can be changed to cover σ, a new test needs to be added in which the parameters involved in σ are assigned the same value in σ and the other parameters are assigned *don't care* values. For example, in Fig. 3, after horizontal growth, combination (P1.1, P2.0, P4.0) has not been covered yet. No existing test can be found such that it can be changed to cover this combination. Thus, we create a new test (P1.1, P2.0, P3.-, P4.0), which is the 9th test in part (c), to cover this combination, where "–" denotes a don't care value. Also note that (P2.0, P3.1, P4.0) is another combination that was not covered either after horizontal growth. This combination can be covered by changing the value of P3 from "-" to 1 in the 9th test.

Now we 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 n be the number of parameters and d the largest domain size of the parameters. The space requirement for π is $O\left(d^{t} \times n^{t-1}\right)$. The time complexity is dominated by horizontal extension. In Section 4, we describe a data structure for storing all the combinations. With this data structure, it takes $O(1)$ time to determine whether or not a t-way combination is already covered, and it takes $O\left(n^{t-1}\right)$ time to determine the number of combinations covered by a test. Thus, it takes $O\left(d \times n^{t-1}\right)$ to determine which value of the new parameter covers the most t-way combinations. As shown in [5] and supported by the experiments in Section 5, the number of tests generated by algorithm IPOG-Test is in $O\left(d^{t} \times \log n\right)$. Thus, the time complexity of horizontal extension, and that of the entire algorithm, is $O\left(d^{t+1} \times n^{t-1} \times \log n\right)$.

$$\begin{array}{ccc} P1 & P2 & P3 \\ \end{array} \quad \begin{array}{ccc|c} P1 & P2 & P3 & P4 \\ \end{array} \quad \begin{array}{cccc} P1 & P2 & P3 & P4 \\ \end{array}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad \left(\begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{array}\right) \quad \left(\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 1 & 1 & 0 & 2 \\ - & 0 & 0 & 2 \\ - & 1 & 1 & 2 \end{array}\right)$$

(a) (b

Figure 3. An illustration of algorithm IPOG-Test

4. FireEye: A T-Way Testing tool

We built a t-way testing tool, called FireEye, which implements the IPOG strategy. FireEye is written in Java and consists of the following major components: (1) *CombinatoricsHelper*, which is a utility class that is 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 algorithm IPOG-Test; (4) *TestGenerator*, which drives the entire test generation process. FireEye also provides a graphic user interface (GUI) to facilitate the use of this tool. The GUI allows the user to create, edit, and inspect system configurations, to set up runtime options, and to view the resulting test sets.

Due to the combinatorial effect, the number of t-way combinations can be large. To enable an efficient implementation, these combinations must be managed carefully. In Section 4.1, we discuss how FireEye computes t-way combinations. In Section 4.2, we describe the data structure for storing these combinations in FireEye.

4.1 Computing T-Way Combinations

In order to cover a new parameter, we first need to compute the set π of t-way combinations involving the new parameter and t-1 parameters that have already been covered (line 5 of Fig. 2). In the following, we consider a more general problem: How can we 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, we generate all possible combinations of n parameters out of m parameters. Second, for each combination of n parameters, we enumerate all possible combinations of values of these n parameters. In the remainder of this paper, we will refer to a *combination of parameters* as a *parameter combination*, and a *combination of parameter 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 both *n*-way parameter combinations, with care given to avoid generating the same combination of parameters in different orders, and *n*-way value combinations. 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[1].

We first discuss how to generate parameter combinations. Center to our approach is the use of parameter vectors. A parameter vector has *m* dimensions, one for each parameter. Consider each parameter vector as representing 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 5 parameters {P0, P1, P2, P3, P4}. Then a parameter vector 10101 represents a parameter combination {P0, P2, P4}. Thus, the problem of generating all the *n*-way parameter combinations is transformed to the problem of generating all the parameter vectors in which the number of 1s 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 1s is not *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 0s, whose numeric value is 0, we can enumerate all the parameter vectors by repeatedly adding 1 until a vector of all 1s 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, we change the third digit from 0 to 1, and set the last two digits to 0. Doing so results in a new vector 10100.

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 1s is exactly *n*. The framework of our approach is similar to that of the naïve approach, except for the following two differences. First, we start from a parameter vector in which the least significant *n* digits are set to 1, instead of the vector of all 0s. For example, let *m* = 5, and *n* = 3. Then, we start from 00111. Note that such a parameter vector is the smallest one, in terms of its numeric value, that consists of three 1s. Second, every time we derive a new parameter vector, we ensure that the number of 1s 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, we find the least significant digit *g* that is 0 and is followed by 1. Then, we change *g* 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, we generate the next parameter vector by changing the third digit from 0 to 1 and the fourth digit from 1 to 0, which produces 01101. 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 1s.
- *Case 2*: If the last digit is 0, we find the least significant digit *g* that is 0 and is followed by 1, which is similar to Case 1. At the same time, we count the number of 1s, say *c*, that appear before *g*. Then, we change *g* from 0 to 1, and set the digits that are less significant than g 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, we generate the next parameter vector 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 1s.

Next we discuss how to enumerate all possible value combinations for each parameter combination. Similar to the way we consider a parameter combination, we consider each value combination 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. Starting from a value combination of all 0s, whose numeric value is 0, we can enumerate all the value combinations by repeatedly adding 1 until we reach a value combination in which the value of each digit is its base

[1] We developed the described approach independently, but were made aware of [15] that provides a similar solution in the review process.

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 P1, P2, and P3, 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. We can add 1 to this combination by incrementing the second digit and by setting the last digit to 0, which results in a new value combination 120.

4.2 Storing T-Way Combinations

In this section, we describe the data structure used by FireEye for storing *t*-way combinations. On the one hand, we want the storage to be as compact as possible. On the other hand, we want to be able to quickly determine whether or not a given combination has been covered, which is the most frequently performed operation in algorithm IPOG-Test.

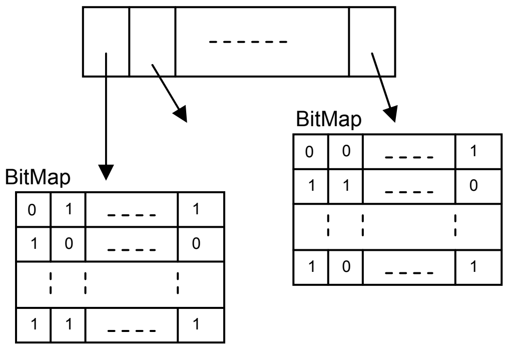

Figure 4. A two-level hierarchy for storing combinations

As shown in Fig. 4, 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. The pointers are indexed in such a way that for a given parameter combination, we can directly compute its index and thus locate the corresponding pointer quickly without having to search through the array. We use an example to illustrate the indexing scheme. Assume that there are 4 parameters, P0, P1, P2, and P3. There are 4 combinations of 3 parameters out of the 4 parameters, and we index them in the following order: (P0, P1, P2), (P0, P1, P3), (P0, P2, P3), and (P1, P2, P3). The index of a given parameter combination (P_i, P_j, P_k) can be computed using the following formula $3\times i+2\times(j-i-1)+(k-j-1)$. For instance, the index of (P0, P2, P3) is $3\times 0+2\times(2-0-1)+(3-2-1)=2$. This formula can be easily generalized to any number of parameters.

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 the corresponding value combination has already been covered. Again, we consider each value combination to represent a numeric value. The numeric value of a value combination is used to index the bit that corresponds to the combination.

In order to determine whether or not a given value combination is covered, we first find the pointer that points to the bitmap to which the value combination belongs. Then, we check the value of the bit corresponding to the combination in the bitmap. Both steps take constant time.

5. Experimental Results

Our experiments have two goals. First, we want to study the growth in the size of the test sets generated by algorithm IPOG-Test, as well as the time taken to produce those test sets, in terms of the strength of coverage, the number of parameters, and the domain size, respectively. Second, we want to compare the performance of FireEye to existing tools, both in terms of the size of the resulting test sets and the time taken to produce these test sets.

To accomplish the first goal, we applied FireEye to three series of system configurations. In the first series, the number of parameters is fixed to 10, the domain size of each parameter is fixed to 5, and the strength of coverage is varied from 2 to 6. In the second series, the strength of coverage is fixed to be 4, the domain size of each parameter is fixed to be 5, and the number of parameters is varied from 5 to 15. In the third series, the strength of coverage is fixed to be 4, the number of parameters is fixed to be 10, and the domain size is varied from 2 to 10.

Tables 1, 2 and 3 show the experimental results for the three series of system configurations, respectively. The columns in the three tables are self-explanatory. Note that the execution times are shown in seconds, and all the results were collected using a laptop running Windows XP with 1.6GHZ CPU and 1GB memory.

In [5], it was shown that the growth in the size of a test set is in $O(d^t \log n)$, where *t* is the strength of coverage, *d* is the domain size, and *n* is the number of parameters. We performed curve fitting analysis on the sizes of the test sets in the three tables. The analysis showed that our experimental results were consistent with the theoretical results. In particular, we note that the number of tests in a *t*-way test grows very quickly as the strength of coverage *t* increases.

To accomplish the second goal, we identified the following existing tools 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 [19]; (2) Jenny, which is from www.burtleburtle.net [20]; (3) TConfig, which is from University of Ottawa [21]; and (4) Test Vector Generator (or TVG), which is from www.SourceForge.com [22]. Based on limited information available in the literature, ITCH implements a combination of several algebraic methods (the details of the combination are not known), and TConfig implements a recursive construction method. Both Jenny and TVG seem to implement a computational method, but the details of their algorithms are not clear. Note that all these tools are written in Java, except for Jenny, which is written in C.

t-way	2	3	4	5	6
Size	48	308	1843	10119	50920
Time	0.11	0.56	6.38	63.8	791.35

Table 1: Results for 10 5-value parameters for 2- to 6-way testing

We applied FireEye and the above tools to a Traffic Collision Avoidance System (TCAS) module. It implements part of an aircraft collision avoidance system specified by the Federal Aviation Administration, and has been used in other studies of software testing [9][12]. The TCAS module has twelve parameters: seven parameters have 2 values, two parameters have three values, one parameter has four values, and two parameters have 10 values. Table 4 shows the sizes of the test sets generated by each tool and the times taken to generate these test sets. The execution times are shown in seconds, if not specified otherwise. The sizes of some test sets are not available, shown as NA, as their construction seems to take an excessive amount of time. In all cases, FireEye has performed better than the other tools, both in terms of the sizes of the test sets and the execution times. In several cases, FireEye has performed substantially better, especially for 5- and 6-way testing. If we compare FireEye to a particular tool, the extent to which FireEye outperformed increases as the strength of coverage increases.

6. Conclusion and Future Work

We consider *t*-way testing to be a very promising testing technique 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 is 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 can be very effective for various types of applications. Kuhn et al. studied actual faults in several industrial applications, showing that all the known faults in these applications are caused by up to 6-way interactions [11]. Finally, test input generation for *t*-way testing can be automated as a push-button feature, which is a key to industrial acceptance.

# of params	5	6	7	8	9	10	11	12	13	14	15
Size	784	1064	1290	1491	1677	1843	1990	2132	2254	2378	2497
Time	0.19	0.45	0.92	1.88	3.58	6.38	10.83	17.52	27.3	41.71	61.26

Table 2: Results for 5 to 15 5-value parameters for 4-way testing

# of values	2	3	4	5	6	7	8	9	10
Size	46	229	649	1843	3808	7061	11993	19098	28985
Time	0.16	0.547	1.8	6.33	16.44	38.61	83.96	168.37	329.36

Table 3: Results for 10 parameters with 2 to 10 values for 4-way testing

t-way	FireEye		ITCH		Jenny		TConfig		TVG	
	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time
2	100	0.8	120	0.73	108	0.001	108	>1 hour	101	2.75
3	400	0.36	2388	1020	413	0.71	472	>12 hour	9158	3.07
4	1361	3.05	1484	5400	1536	3.54	1476	>21 hour	64696	127
5	4219	18.41	NA	>1 day	4580	43.54	NA	>1 day	313056	1549
6	10919	65.03	NA	>1 day	11625	470	NA	>1 day	1070048	12600

Table 4: Results of different tools for the TCAS configuration

We are continuing our work in the following directions. First, the IPOG strategy needs to explicitly enumerate all possible combinations. When the number of combinations is large, explicit enumeration can be prohibitive. We are developing techniques to reduce the number of combinations that are enumerated. Second, we are extending algorithm IPOG-Test 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 meaningful from the domain semantics. Finally, *t*-way testing can generate a large number of tests, which makes it impractical to manually execute the tests and evaluate their results. We plan to integrate our test generation tool with other tools to automate the entire testing process, i.e., including test generation, test execution, and test evaluation.

Acknowledgement

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

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

References

[1] K. A., Bush, "Orthogonal arrays of index unity," Annals of Mathematical Statistics, 23 (1952), 426-434.

[2] R. Bryce, A Deterministic Density Algorithm for Pairwise Interaction Coverage, Proceedings of the International Conference on Software Engineering (SE 2004). Innsbruck, Austria, pp. 245-252.

[3] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998.

[4] D. M. Cohen, S. R. Dalal, J. Parelius, G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, Vol. 13, No. 5, pp. 83-87, September 1996.

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," IEEE Transactions on Software Engineering, 23:7, 1997.

[6] M. B. Cohen, C. J. Colbourn, P. B. Gibbons and W. B. Mugridge, "Constructing test suites for interaction testing," In Proc. of the Intl. Conf. on Software Engineering, (ICSE 2003), 2003, pp. 38-48, Portland.

[7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based testing in practice," in Proc. of the Intl. Conf. on Software Engineering, (ICSE), 1999, pp. 285–294.

[8] Alan Hartman, Leonid Raskin, "Problems and algorithms for covering arrays," Discrete Mathematics 284(1-3): 149-156 (2004)

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. Sixteenth Internat. Conf. On Software Engineering*, pp. 191–200, May 1994.

[10] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," Proceedings of the 27th NASA/IEEE Software Engineering Workshop, NASA Goddard Space Flight Center, December 2002.

[11] D. R. Kuhn, D. Wallace, A. Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions on Software Engineering, June 2004, Vol. 30, No. 6.

[12] D. R. Kuhn, V. Okun, "Pseudo-exhaustive Testing For Software," 30th NASA/IEEE Software Engineering Workshop, April 25-27, 2006.

[13] Y. Lei and K. C. Tai , "In-parameter-order: a test generation strategy for pairwise testing," Proceedings of 3rd IEEE Intl. Conf. on High-Assurance Systems Engineering Symposium, 1998, pp. 254-261.

[14] R. Mandl, "Orthogonal Latin squares: an application of experiment design to compiler testing," Communications of the ACM, v.28 n.10, p.1054-1058, Oct. 1985.

[15] D. Stanton and D. White, Constructive Combinatorics, Springer, 1986.

[16] K. C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," IEEE Transactions on Software Engineering, 2002, Vol. 28, No. 1.

[17] Y. W. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," Proceedings of IEEE Aerospace Conference, 2000, pp. 431-437.

[18] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. In Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE), White Plains, New York, 1996.

[19] ITCH, http://www.alphaworks.ibm.com/tech/whitch.

[20] Jenny, http://www.burtleburtle.net/bob/math/.

[21] TConfig, http://www.site.uottawa.ca/~awilliam/.

[22] TVG, http://sourceforge.net/projects/tvg/.