3-way Interaction Testing using the Tree Strategy

Mohammad F. J. Klaiba, Mohammad Subhi Al-batah and Rashad J. Rasras

Abstract

Failures of hardware and software systems are often caused due to unexpected interactions among system components. The number of tests that needs to be performed in order to test all possible combinations of interactions can be exorbitant even for medium sized projects. To bring a balance between exhaustive testing and lack of testing, researchers have adopted pairwise testing which promises the testing of all pairwise combinatorial interactions between input components. This paper enhances the previous strategy, “A Tree Based Strategy for Test Data Generation and Cost Calculation” to go beyond pairwise testing. The new strategy can support 3-way combinatorial interaction testing.

Keywords: Combinatorial interaction testing, Software testing, Hardware testing, 3-way testing.

1. Introduction

Failures of hardware and software systems are often caused due to unexpected interactions among system components. The main reason for failure is the lack of proper testing. A complete test requires testing all possible combinations of interactions, which can be exorbitant even for medium sized projects due to the huge number of combinations (Combinatorial explosion problem). Thus, to bring a balance between exhaustive testing and lack of
testing, combinatorial interaction testing [22-41] has demonstrated to be an effective technique to achieve reduction of test suite size.

There are a number of strategies proposed in literature for test suite generation of combinatorial interaction testing. Most of these strategies work only for pairwise combinatorial software interaction testing and a few others have been extended to work for higher strength testing. Combinatorial interaction testing strategies could be broadly classified into two types [20] based on the approach that is used to solve the problem. They are:

- Algebraic strategies
- Computational strategies

Algebraic approaches have pre-defined rules to compute test suites directly from mathematical functions [20]. On a contrary, computational approaches use search technique to search the combinations space to generate the test cases until all T-way combinations of interactions to be covered. A number of researches have worked in this field and have adopted either the computational or algebraic approaches.

The classification of strategies used for combinatorial software testing has been further extended by Grindal et al. [21, 22] into three main categories based on the randomness of the implemented solution. They are:

- Deterministic strategies
- Non-deterministic strategies
- Compound strategies

A deterministic strategy is one which has the property that it produces the same test suite for every execution. A non-deterministic strategy on the other hand has the property that for every execution, there is always a randomly generated combination suite to cover all the required T-way combinations. In a compound strategy two or more combination of strategies are used together.

The Automatic Efficient Test Generator or AETG [4, 24] and its variant mAETG [20] employ the computational approach. This approach uses ‘Greedy technique’ to construct test cases based on the criteria that every test case covers as many uncovered combinations as possible. The AETG uses a random search algorithm and hence the test cases are generated in a highly non-deterministic fashion [25]. Other variants of AETG use the Genetic Algorithm, Ant Colony Algorithm [21].

In Genetic algorithm [21] an initial population of individuals (test cases) are created and then the fitness of the created individuals is calculated. Then the individual selection methods are applied to discard the unfit individuals. The genetic operators such as crossover and mutation are applied to the selected individuals and this continues until we evolve a set of best individuals or the stopping criteria is attained. Thus this approach follows a non deterministic methodology similar to the Ant Colony Algorithm [21] in which each path from start to end point is associated with a candidate solution. The candidate solution is the amount of pheromone deposited on each edge of the path followed by an ant, when it reaches the end point. When an ant has to choose among the different edges, it would choose the edge with a large amount of pheromone with higher probability thus leading to better results. In some cases, these algorithms give optimal solution than original AETG.

The In-Parameter-Order [14] or IPO Strategy for pairwise testing starts constructing the test cases by considering the first two parameters, then uses a horizontal growth strategy which extends to cover the third, fourth, fifth etc. until all the parameters are considered. Further it adopts a vertical growth strategy which helps in covering all the pairs that are not covered, until all the pairs in the covering array are covered. Thus this approach generates the test cases in a deterministic fashion. Covering one parameter at a time gives a lower order of complexity to this strategy than AETG. The IPOG [6, 23] strategy extends IPO, so that IPOG can generate test suite supporting T-way combinatorial interactions. The IRPS Strategy [27] uses the computational approach and so generates all pairs and stores them in a linked list and then searches the list to arrive at the best set of test cases in a deterministic fashion.
The G2Way [1] uses a computational and deterministic strategy. It adopts a backtracking strategy to generate the test cases. The main algorithms that form the G2Way strategy consist of the parser algorithm, the 2-way combination generation algorithm, the backtracking algorithm, and the executor algorithm. The parser algorithm will load the parameter and values to be used by the 2-way combination generation algorithm which generates the 2-way covering array. Exploiting the result generated by the combination generation algorithm, the backtracking algorithm generates the 2-way test sets in two phases. In the first phase, the sets generated by the combination generation algorithm are merged together to form complete test suites. In the second phase, all the test sets in the generated test suite are checked to ensure that all the combinations in the covering array are covered. GTWay adopts the same strategies as that of G2Way but generates test suites for general and high T-way combinatorial interaction strengths.

The TConfig [17] uses a deterministic approach to construct test suites for T-way testing. It uses a recursive algorithm for pairwise interaction testing and a version of IPO for T-way testing. TConfig was mainly developed for pairwise interaction test suite generation by applying the theory of orthogonal latin squares from balanced statistical experiments. Jenny [18] is a tool similar to AETG, which first covers single features (one way interaction), then pairs (2-way interaction) of features, then triples (3-way interaction), and so forth up to the n-tuples requested by the user. During each pass it checks whether the existing tests cover all tuples, and if not, make a list of uncovered tuples and add more tests until all tuples are covered. It tries to find test cases that obey the restrictions and cover a lot of new tuples. Any tuple that it can't cover no matter how hard it tries without disobeying some restriction, it says it can't cover it, and adds it to the list of restrictions. Thus it uses a computational and deterministic approach for test suite generation.

WHITCH is IBM’s Intelligent Test Case Handler. With the given coverage properties it uses combinatorial algorithms to construct test suites over large parameter spaces. TVG [19] is a free tool that is built based on model based techniques. It combines both behaviour and data modelling techniques. The behaviour modelling allows the testers to capture important high level scenarios to test. A data model is then created at a level of sophistication according to the importance of each test scenario.

Other researchers have adopted heuristic search techniques [26] such as Hill climbing, Simulated Annealing, Tabu search, Great Flood etc. All of these search strategies have the same goal as to maximize the number of tuples covered in a test. It initially uses greedy algorithm to choose each test and then it is modified using local search. These Heuristic search techniques predict the known test set in advance in contrast to AETG and IPO which builds the test set from the scratch. However, there is no guarantee that the test set produced by Heuristic techniques are the most optimum. The AETG or IPO takes longer time to complete when compared to the Heuristic techniques. Although some work has been done in the past by researchers, test suite generation for combinatorial interaction testing still remains a research area and NP complete problem that needs exploration.

Testing all pairwise (2-way) interactions between input components ensure the detection of 50 – 97 percent of faults [7], [8-11], [13-19]. Although using pairwise testing gives a good percentage of reduction in fault coverage, empirical studies show that pairwise testing is not sufficient enough for highly interactive systems [12]. Therefore, there is a need to extend the level of testing strength to a higher degree.

Therefore, based on the above argument, this work extends our previous strategy “A Tree Based Strategy for Test Data Generation and Cost Calculation” to go beyond pairwise combinatorial interaction testing, and extend the level of the testing strength to 3-way using two algorithms, a tree generation algorithm which generates the test cases and an iterative cost calculation algorithm which enables a minimum 3-way test data generation.
2. Methodology

Here in this paper we adopted computational strategy which based on a searching method to create test suite. The proposed strategy starts by constructing the test-tree based on the input parameters and values. The algorithm then constructs the covering array, for all possible combinations of input variables. After which the cost array corresponding to the number of test cases (or leaf nodes) is created and initialized to some high value. Then, the cost calculation begins. The algorithm first calculates the maximum cost or maximum number of pairs that can be covered by any test case for the given set of parameters and values. Then it iterates to calculate the cost of each and every leaf node which represents the test cases, in a sequential order. The cost of any leaf node or test case is equal to the number of multi-way that it covers in the covering array.

Once it reaches a leaf node with the maximum cost, it deletes this leaf node from the list of leaf nodes generated by the test-tree and includes this node or test case into the new list which holds all the test cases that are to be included in the test suite. It also deletes all the pairs that this test case has covered from the covering array.

To illustrate the concept consider a simple system with parameters and values as shown below:

- Parameter A has two values A1 and A2
- Parameter B has one value B1
- Parameter C has three values C1, C2 and C3
- Parameter D has two values D1 and D2

![Figure 1 Test-Tree Construction](image)

Fig. 1 above shows how the test-tree would be constructed. The test cases generated by the test-tree are stored in the list T in a sequential order i.e. T1(A1,B1,C1,D1), T2(A1,B1,C1,D2), T3(A1,B1,C2,D1), T4(A1,B1,C2,D2),...
T5(A1,B1,C3,D1), T6(A1,B1,C3,D2), T7(A2,B1,C1,D1), T8(A2,B1,C1,D2), T9(A2,B1,C2,D1),
T10(A2,B1,C2,D2), T11(A2,B1,C3,D1) and T12 (A2,B1,C3,D2).

The algorithm then constructs the covering array, for all possible multi-way combinations of input variables. Table 1 shows the covering array for pairwise combinations i.e. [A & B], [A & C], [A & D], [B & C], [B & D] and [C & D].

<table>
<thead>
<tr>
<th>A with B</th>
<th>A with C</th>
<th>A with D</th>
<th>B with C</th>
<th>B with D</th>
<th>C with D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1,B1</td>
<td>A1,C1</td>
<td>A1, D1</td>
<td>B1,C1</td>
<td>B1, D1</td>
<td>C1, D1</td>
</tr>
<tr>
<td>A2,B1</td>
<td>A1,C2</td>
<td>A1, D2</td>
<td>B1,C2</td>
<td>B1, D2</td>
<td>C1, D2</td>
</tr>
<tr>
<td>A1,C3</td>
<td>A2, D1</td>
<td>B1,C3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2,C1</td>
<td>A2, D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2,C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2,C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The covering array for the above example has 23 pairwise interactions which have to be covered by any test suite generated, to enable a complete pairwise interaction testing of the system.

3-way covering array can be also generated as in table 2:

<table>
<thead>
<tr>
<th>A, B, C</th>
<th>A, B, D</th>
<th>A, C, D</th>
<th>B, C, D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1, B1, C1</td>
<td>A1, B1, D1</td>
<td>A1, C1, D1</td>
<td>B1, C1, D1</td>
</tr>
<tr>
<td>A1, B1, C2</td>
<td>A1, B1, D2</td>
<td>A1, C1, D2</td>
<td>B1, C1, D2</td>
</tr>
<tr>
<td>A1, B1, C3</td>
<td>A2, B1, D1</td>
<td>A1, C2, D1</td>
<td>B1, C2, D1</td>
</tr>
<tr>
<td>A2, B1, C1</td>
<td>A2, B1, D2</td>
<td>A1, C2, D2</td>
<td>B1, C2, D2</td>
</tr>
<tr>
<td>A2, B1, C2</td>
<td>A1, C3, D1</td>
<td>B1, C3, D1</td>
<td></td>
</tr>
<tr>
<td>A2, B1, C3</td>
<td>A1, C3, D2</td>
<td>B1, C3, D2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C1, D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C1, D2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C2, D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C2, D2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C3, D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A2, C3, D2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the test-tree construction is over we have all the test cases generated (shown in figure 1), and the covering arrays (shown in table 1 and 2). Then the cost calculation begins: The algorithm first calculates the maximum cost or maximum number of pairs that can be covered by any test case for the given set of
parameters and values. Then it iterates to calculate the cost of each and every leaf node which represents the test cases, in a sequential order. The cost of any leaf node or test case is equal to the number of pairs that it covers in the covering array.

3. Results

To evaluate the efficiency of the proposed strategy TBGCC-3 for 3-way test data generation, we consider a system with 5 parameters, each parameter has 2-valued. Table 3 shows the exhaustive number of test cases, test suite size, and the percentage of test size reduction.

<table>
<thead>
<tr>
<th></th>
<th>Exhaustive number of test cases</th>
<th>TBGCC-3 Test suite size</th>
<th>Reduction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way</td>
<td>32</td>
<td>6</td>
<td>81.25%</td>
</tr>
<tr>
<td>3-way</td>
<td>32</td>
<td>12</td>
<td>62.5%</td>
</tr>
</tbody>
</table>

The efficiency of our strategy will be also compared with available software testing strategies in terms of test size reduction. The following strategies and tools can support pairwise and/or higher interaction: AllPairs [16], TConfig [17], Jenny [18], TVG [19], GTWay [1] tool. We consider the same system configuration as described above.

<table>
<thead>
<tr>
<th>System</th>
<th>TConfig</th>
<th>Jenny</th>
<th>TVG</th>
<th>ALL Pairs</th>
<th>G2Way</th>
<th>TBGCC-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-way</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3-way</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>NS</td>
<td>NS</td>
<td>12</td>
</tr>
</tbody>
</table>

NS – Not Supported
4. Discussion

In this paper we extend and improve our previous strategy, “A Tree Based Strategy for Test Data Generation and Cost Calculation” to support 3-ways testing strength interactions. The proposed strategy is based on two algorithms. A tree construction algorithm which constructs the possible test cases and an iterative cost calculation algorithm that constructs efficient 2-way and 3-way test suites which cover all possible combinatorial interactions between input components.

Table 3 reveals that the proposed strategy works well for different test strength (2 and 3ways) values, and can produce an efficient and reduced test suite size.

Tables 4 displays the comparison of test suite size generated by our strategy with other strategies. The minimum test suite size is highlighted. Our strategy (TBGCC-3) has been one of the best results for most of the test.

Empirical results in Section 3 shows that our strategy (TBGCC-3) is an efficient strategy in test size reduction and can generate highly reduced test suites.

References