ICIT 2015 The 7th International Conference on Information Technology
doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

Anatomy of the Tree Based Strategy for High Strength Interaction Testing

Mohammad F. J. Klaib
Computer Science Department
College of Computer Sciences and Engineering
Taibah University
Madina, Kingdom of Saudi Arabia
Email: mklaib@taibahu.edu.sa
mom_klaib@yahoo.com

***Abstract*—The amount of resources consumed for a complete and exhaustive testing becomes unreasonable and unaffordable. While it is vital to assure the quality and the reliability of any system, it is impossible to do an exhaustive testing due to the huge number of possible combinations. To bring a balance between exhaustive testing and lack of testing combinatorial interactions testing has been adopted. Although it is stated in literature that a complete pairwise interaction testing ensures the detection of 50–97 percent of faults, it is not sufficient to stop with pairwise testing alone for highly interactive systems. Therefore, there is a need to extend the level of testing for a general multi way combinatorial interactions testing. This paper enhanced the previous strategies "A tree based strategy for test data generation and cost calculation" and "3-way interaction testing using the tree strategy" to support a general multi-way combinatorial interaction testing involving uniform and non uniform parametric values. In this strategy, two algorithms have been adopted; a tree construction algorithm which constructs the possible test cases and an iterative cost calculation algorithm that constructs efficient multi-way test suites which cover all parameter interactions between input components. Both algorithms are presented in details.**

Keywords— Software testing, Hardware testing, Multi-way testing

I. INTRODUCTION

Testing [1] is an activity aims to evaluate the attributes or capabilities of software or hardware products, and determines if the products have met their requirements. Testing in general is a very important phase of the development cycle for both software and hardware products [2-5]. Testing helps to reveal the hidden problems in the product, which otherwise goes unnoticed providing a false sense of well-being. It is said to cover 40 to 50 percent of the development cost and resources [6,7]. Although important to quality and widely deployed by programmers and testers, testing still remains an art. A good set of test data is one that has a high chance of uncovering previously unknown errors at a faster pace. For a successful test run of a system, we need to construct a good set of test data covering all interactions among system components [34-39].

Failures of hardware and software systems are often caused due to unexpected interactions among system components. The failure of any system may be catastrophic that we may lose very important data or fortunes or sometimes even lives [7,8]. 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).

Combinatorial Explosion – All products are built with basic elements which interact with one another by means of predefined combination rules. As the number of classes of elements increases, the number of interactions between the elements also increases exponentially [9-11] which leads to the problem of combinatorial explosion. Thus, combinatorial explosion [21,22] occurs when a huge number of possible combinations are produced by increasing the number of entities or elements, which have to interact with one another for successful functioning of a product.

To gain a better understanding of this problem, we consider a simple example of testing a 16-1 multiplexer. A multiplexer or mux is a device that selects one of many analog or digital input signals and forwards the selected input to a single output line. A multiplexer of 2^n inputs has n select lines, which are used to select which input line will be directed the output. Fig. 1 below shows a black box of 16-1 multiplexer. In order to exhaustively test such a multiplexer, there are 2^{20} (i.e. 1048576) combinations of tests that needs to be performed. If the time required for one test to be executed is 5 minutes, then it would take nearly 10 years for a complete test to be done. Thus, the amount of resources consumed for a complete and exhaustive testing of the system becomes unreasonable and unaffordable [12,13]. While it is vital to assure the quality and the reliability of the system, it is impossible to do an exhaustive testing due to the combinatorial explosion problem. Therefore, it is very clear that combinatorial explosion is a serious and critical issue that all software and hardware testers face.

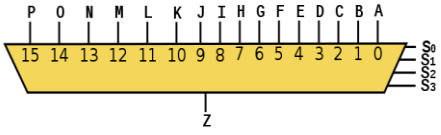

Fig 1 16-1 Multiplexer

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

Thus, to bring a balance between exhaustive testing and lack of testing, combinatorial interaction testing [14-16] has demonstrated to be an effective technique to achieve reduction of test suite size, thus relieving the problem of combinatorial explosion. Combinatorial interaction testing, samples the systems input space and produces a set of factor-value bindings that typically cover all possible pairs or multi-way combinations of factor-values, thereby achieving a high-degree of coverage and fault detection [17].

Testing all pairwise (2-way) interactions between input components ensure the detection of 50 – 97 percent of faults [17-19], [24-30]. 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 [23]. Therefore, there is a need to extend the level of testing to support higher muti-way combinatorial interactions, which requires every combination of any T parameter values to be covered by at least one test case, where T is referred to as the strength of coverage. Constructing a minimum test set for multi-way combinatorial interaction is still a NP complete problem [19,20] and there is no strategy that can claim that it has the best generated test suite size for all cases and systems.

Therefore, based on the above argument, this new work extends our previous strategy "A Tree Based Strategy for Test Data Generation and Cost Calculation" to go beyond pairwise combinatorial interaction testing involving uniform and non-uniform parametric values. We have two algorithms, a tree generation algorithm which generates the test cases and an iterative cost calculation algorithm which enables a minimum multi-way test data generation. The remainder of this paper is organized as follows. Section 2 presents the related work. In Section 3, the proposed tree generation and the iterative cost calculation strategies are illustrated and the correctness of both strategies have been proved with an example. Section 4 provides the conclusion.

II. RELATED WORK

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 T-way testing. Combinatorial interaction testing strategies could be broadly classified into two types [31] 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 [31]. 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. [19, 20] 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 [9, 14] and its variant mAETG [31] 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 [22]. Other variants of AETG use the Genetic Algorithm, Ant Colony Algorithm [20].

In Genetic algorithm [20] 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 [20] 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 [25] 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 [8, 16] strategy extends IPO, so that IPOG can generate test suite supporting T-way combinatorial interactions. The IRPS Strategy [33] 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.

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

The G2Way [13] 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 [28] 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 [29] 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 [30] is a free tool that is built based on model based techniques. It combines both behavior and data modelling techniques. The behavior 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.

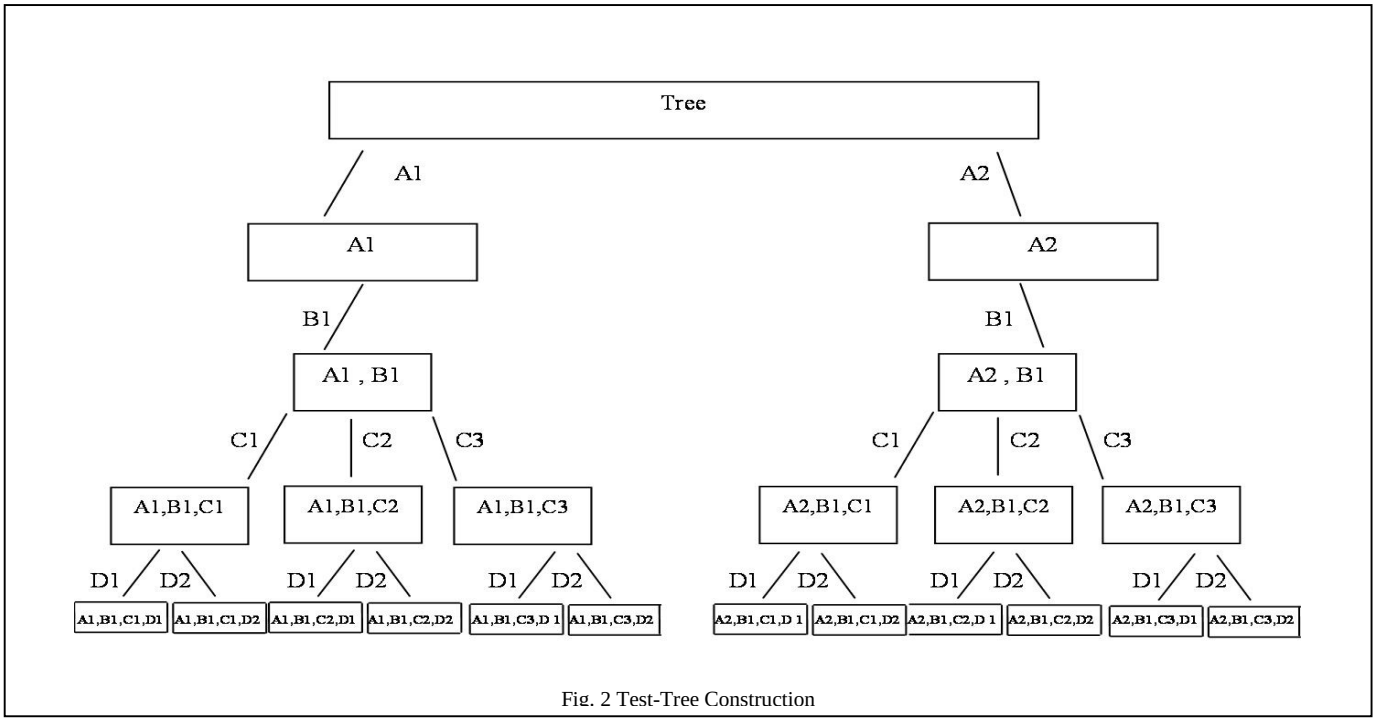

Fig. 2 Test-Tree Construction

Other researchers have adopted heuristic search techniques [32] 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.

III. THE PROPOSED STRATEGY

The proposed strategy starts by constructing the test-tree based on the input parameters and values. Then it constructs the covering array, which includes all possible multi-way combinations of input variables. In order to construct the test-

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

tree it considers one parameter at a time until all the values of all the parameters are considered. 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

We have given the illustration for minimum test suite construction of 2-way and 3-way combinatorial interactions using our algorithm, for the example in Fig. 2 above, which depicts the system mentioned. The tree generation algorithm starts by constructing the test-tree. It uses the values of the first parameter to construct the base branches of the test-tree. Then it uses all the values of the second parameter for the next level and then the third and so on. Thus, the tree is constructed iteratively until all the parameters are considered. As a result we get all possible test cases generated for all the parameters by considering all its values.

Fig. 2 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 2-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]. 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.

Once the test-tree construction is over we have all the test cases generated. The next step generates the covering array, after which the cost array corresponding to the number of test cases (or leaf nodes) is created and initialised 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 pairs that it covers in the covering array.

TABLE 1. PAIRWISE INTERACTION COVERING ARRAY

A with B	A with C	A with D	B with C	B with D	C with D
A1,B1	A1,C1	A1, D1	B1,C1	B1, D1	C1, D1
A2,B1	A1,C2	A1, D2	B1,C2	B1, D2	C1, D2
	A1,C3	A2, D1	B1,C3		C2, D1
	A2,C1	A2, D2			C2, D2
	A2,C2				C3, D1
	A2,C3				C3, D2

TABLE2. GENERATED TEST SUITE FOR PAIRWISE COMBINATORIAL INTERACTION

Test Case No.	Test Case	Iteration	Max Weight	Covered pairs
T1	A1,B1,C1,D1	1	6	[A1,B1][A1,C1][A1,D1] [B1,C1][B1,D1][C1,D1]
T10	A2,B1,C2,D2	1	6	[A2,B1][A2,C2][A2,D2] [B1,C2][B1,D2][C2,D2]
T6	A1,B1,C3,D2	2	4	[A1,C3][A1,D2] [B1,C3][C3,D2]
T11	A2,B1,C3,D1	3	3	[A2,C3] [A2,D1] [C3,D1]
T3	A1,B1,C2,D1	4	2	[A1,C2] [C2,D1]
T8	A2,B1,C1,D2	4	2	[A2,C1] [C1,D2]

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 i.e. T and includes this node or test case into the new list T' 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. In the above example, when the first iteration begins, the first leaf node (A1,B1,C1,D1) will be deleted from T and added to T' since it has a cost equals the maximum cost 6 and the six pairs covered by it ([A1,B1], [A1,C1], [A1,D1], [B1,C1], [B1,D1] and [C1,D1]) will be deleted from the covering array. Thus, the first leaf node (or test case) generated by the test-tree will always have the maximum cost and is said to be included in T' by default for any system.

The algorithm will then continue calculating the cost of all the leaf nodes in a sequential order and includes the ones having the maximum cost. If all the pairs in the covering array are covered then the algorithm stops else it goes to the second iteration. In the second iteration, the maximum cost value (Wmax) will be decreased by one and the next set of best test cases (i.e. test cases that can cover the new Wmax number of the covering array. Thus, the algorithm continues until all the pairs in the covering array are covered. For the above example all the test cases which are included in the test suite are identified in four iterations and there are six such test cases.

Table 2 shows how the cost calculation algorithm works iteratively to generate the test suite. Table 2 also shows the order in which the various test cases are actually included in the test suite. Thus, all the 23 pairs generated for covering all pairwise

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

interactions as shown in Table 1, has been covered by the test cases generated by our algorithm as shown in the fifth column of the Table2. Thus this proves the correctness of our strategy in generating pairwise test suite. We have also proved that our algorithm is efficient in achieving a good reduction in the number of test cases. The exhaustive number of test cases is 12 and we have generated 6 test cases which covers all the pairs in the covering array thus achieving a 50% reduction in this case.

i.e. [A, B, C], [A, B, D], [A, C, D] and [B, C, D], for the example in Fig. 2. The covering array for the above example has 28 combinations of 3-ways interactions which have to be covered in the final test suite. Table 4 shows how the cost calculation algorithm works iteratively to generate the test suite. It also shows the order in which the various test cases are actually included in the test suite. All the 28 3-way combinations in the covering array have been covered by our algorithm. Thus, the correctness of our strategy for 3-way interaction coverage has been proved.

A. Test—Tr ee *Construction* Algorithm

The tree generation strategy thus provides the following advantages:

TABLE3. 3-WAY INTERACTION COVERING ARRAY

A, B, C	A, B, D	A, C, D	B, C, D
A1, B1, C1	A1, B1, D1	A1, C1, D1	B1,C1, D1
A1, B1,C2	A1, B1,D2	A1, C1, D2	B1,C1, D2
A1, B1,C3	A2, B1,D1	A1, C2, D1	B1,C2, D1
A2, B1,C1	A2, B1,D2	A1, C2, D2	B1,C2, D2
A2, B1,C2		A1, C3, D1	B1,C3, D1
A2, B1,C3		A1, C3, D2	B1,C3, D2
		A2, C1, D1	
		A2, C1, D2	
		A2, C2, D1	
		A2, C2, D2	
		A2, C3, D1	
		A2, C3, D2	

TABLE4. GENERATED TEST SUITE FOR 3-WAY COMBINATORIAL INTERACTION

Test Case No.	Test Case	Iteration	Max Weight	Covered pairs
T1	A1,B1,C1,D1	1	4	[A1,B1,C1][A1,B1,D1][A1,C1,D1][B1,C1,D1]
T4	A1,B1,C2,D2	1	4	[A1,B1,C2][A1,B1,D2][A1,C2,D2][B1,C2,D2]
T8	A2,B1,C1,D2	1	4	[A2,B1,C1][A2,B1,D2][A2,C1,D2][B1,C1,D2]
T9	A2,B1,C2,D1	1	4	[A2,B1,C2][A2,B1,D1][A2,C2,D1][B1,C2,D1]
T5	A1,B1,C3,D1	2	3	[A1,B1,C3][A1,C3,D1][B1,C3,D1]
T12	A2,B1,C3,D2	2	3	[A2,B1,C3][A2,C3,D2][B1,C3,D2]
T2	A1,B1,C1,D2	3	1	[A1,C1,D2]
T3	A1,B1,C2,D1	3	1	[A1,C2,D1]
T6	A1,B1,C3,D2	3	1	[A1,C3,D2]
T7	A2,B1,C1,D1	3	1	[A2,C1,D1]
T10	A2,B1,C2,D2	3	1	[A2,C2,D2]
T11	A2,B1,C3,D1	3	1	[A2,C3,D1]

After the pairwise test suite is generated, we move to the next iteration where we generate the test suite for 3-way combinatorial interactions and so on and so forth until (n-1) way combinatorial interaction test suite is generated. To illustrate the 3-way test suite generation, again the whole process starts by constructing the 3-way covering array and the iterative cost calculation of the test cases in a sequential order as explained before. Table 3 shows the covering array for 3-way combination

- A systematic method whereby all possible test cases are generated in order.

- The above procedure works well for both parameters with uniform and non-uniform values. Therefore all parameters can have different or same values as any real time system to be tested would have.

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

- The algorithm generates only a set of leaf nodes at every stage, although it appears as if the entire tree gets generated in order to minimise the space requirements. Therefore we only have a list of leaf nodes (or test cases) when the algorithm ends.

The example tree shown in Fig. 2 explains how the test cases are constructed manually. In reality we may need only the leaf nodes and all the intermediate nodes are not used. Therefore in order to increase the efficiency and to minimise memory allocation, we have constructed the tree shown in Fig. 2 using the proposed tree generation algorithm, which constructs the tree by minimising the number of nodes and by giving importance to only the leaf nodes at every stage.

Strategy Tree generation

Begin

for the first parameter p1

T = {(v_1), (v_2)……(v_j) / v_1, v_2 and v_j are values of p1 and are sequentially connected}

If n=1 then stop;

f or the remaining parameters

For parameter p_i, i = 2,3,……n
do
Begin
For each Test (v_1, v_2,……, v_j) in T do
Begin

Replicate the Test as many times as (the number of values of p_i – 1)
Add all the replicated nodes sequentially after the current original test node and before the other Test nodes
For each value in p_i do
Begin
Append the original with v_1 and all the replicated tests with (v_2, v_3……v_{i-1}, v_i) Where v_i is a value of p_i and each of which is considered in order

End
End
End

End

Fig. 3 Test Tree Construction Algorithm

Therefore, at each stage or iteration we look at the leaf nodes of the tree and generate the next level nodes by considering all the values of the current parameter to generate the new set of nodes. The new set of leaf nodes from an already existing set is calculated using a replication strategy. If the existing set of leaf nodes is Esoln, new set of leaf nodes is Nsoln and the number of values of the parameter under consideration is n. Then,

$$Nsoln = Esoln * n \quad (1)$$

Assume there are 6 leaf nodes in existing set (i.e. Esoln= 6), and the next parameter to be considered has 2 values (i.e. n=2). Then based on Equation 1 the new list of nodes will have 12 new leaf nodes as a result (i.e. Nsoln=12). Therefore at every stage of tree construction, the algorithm considers each and every existing leaf node separately and calculates the number of times this particular node needs to be replicated in order to get the new set of leaf nodes with the formulae given below:

$$\text{The number of values of } p_i - 1 \quad (2)$$

Where pi – is the ith parameter under consideration for constructing the new set of leaf nodes and i=1,2,….N – the number of parameters.

In Figure 2, consider the existing nodes (A1, B1) and (A2, B1). To construct the next level of nodes the parameter under consideration is C which has values C1, C2 and C3. Therefore, the node (A1, B1) needs to be replicated twice. Now we will have three (A1, B1) nodes to which C1 is added to the first, C2 is added to the second and C3 is added to the third and then the two replicated nodes are included in the list of leaf nodes after the original node and before the node (A2, B1). The same is done to (A2, B1). It is replicated twice and hence we have three of it (one original and two replicated nodes). Now C1 is added to the first (original node), C2 is added to the second (replicated node) and C3 is added to the third (replicated node). Thus we have (A2, B1, C1), (A2, B1, C2) and (A2, B1, C3). The same process is

doi:10.15849/icit.2015.0120 © ICIT 2015 (http://icit.zuj.edu.jo/ICIT15)

done to construct the test-tree until all the parameters are considered. Thus, once the list of leaf nodes is generated by considering all the values of all the parameters, we proceed to the next strategy of iterative cost calculation to construct the test suite.

B. Iterative Cost Calculation Strategy

In Figure 4, the outer loop iterates N-2 times through the list of test cases to generate N-2 test suites, i.e. 2-way, 3-way, 4-way etc. until (N-1) way. For every T-way (i.e. 2-way, 3-way, 4-way etc. until (N-1) way) test suite to be generated, the inner loop of the algorithm iterates until all the combinations of the corresponding T-way covering array are covered. During each iteration, all the test cases with the maximum cost (Wmax) will be included in the test suite. Thus the algorithm guarantees identifying a minimum set of test cases for parameters with uniform and non-uniform values.

```
Strategy T-Way Test Suite Generation by Iterative Cost Calculation

Begin
    Temp = T
     For i= 2 to N-1 do
        Begin
              Generate i-way covering array for the given parameters.
              Create a cost array corresponding to the T list.
              Initialise each element in the cost array to infinity (highest value).
              Let T' be an empty set.
              Wmax = N!/((i!)*((N-i)!))   // N – is the number of   parameters
              While (covering array is not empty) do
                        Begin
                             For each Test Tj in T do // j =1, 2,…n    where there are n test cases in T
                                  Begin
 Mark all i-way combinations that Tj covers in the covering array
 Cost[Tj] = The number of i-way combinations covered in the covering array
 If (Cost[Tj] = = Wmax)
    Begin
                                            T' = T' U Tj
     Delete Tj from T and its corresponding cost from the cost array
                                          Delete all the marked pairs from the covering array
                                        End
                                    Unmark all i-way combinations marked in the covering array
                                End
                                 Wmax--;
                        End
                    Store the minimum i-way Test set T' that is generated
                   T = Temp
          End
End
```

Fig. 4 Cost Calculation Algorithm

IV. CONCLUSION

In this paper we extend and improve our previous strategy, "A Tree Based Strategy for Test Data Generation and Cost Calculation" to support higher 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 multi-way test suites which cover all possible combinatorial interactions between input components. The correctness of the proposed strategy has been proved in Tables 2 and 4.

REFERENCES

[1] C. Kaner, "Exploratory Testing," in Proc. of the Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, 2006.

[2] R. Bryce, C. J. Colbourn, and M. B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Tests," in Proc. of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 146-155.

[3] F. F. Tsui and O. Karam, Essentials of Software Engineering. Massachusetts, USA: Jones and Bartlett Publishers, 2007.

[4] L. G. Hernandez, J. T Jimenez, N. R Valdez, J. B. Rios, "A Post-optimization Strategy for Combinatorial Testing: Test Suite Reduction through the Identification of Wild Cards and Merge of Rows", in Advances in Computational Intelligence Lecture Notes in Computer Science vol. 7630, 2013pp 127-138.

[5] J. Zhou, J. Liu, J. Wu, and G. Zhong, "A Latent Implementation Error Detection Method for Software Validation", Journal of Applied Mathematics. 2013pp 1-10.

[6] B. Beizer, Software Testing Techniques, 2 ed. NY: Thomson Computer Press, 1990.

[7] M.F.J. Klaib, K.Z. Zamli, N.A.M. Isa, M.I. Younis, "G2Way A Backtracking Strategy for Pairwise Test Data Generation" Software Engineering Conference, 2008. APSEC 08, 2008, pp. 463 – 470.

[8] K. Z. Zamli, M. F.J. Klaib, M. I. Younis, N. A. M. Isa, R. Abdullah, "Design and implementation of a t-way test data generation strategy with

automated execution tool support". Journal of Information Sciences. vol 181(9), 2011, pp. 1741‑1758.

[9] S. Khatun, K. F. Rabbi, C. Y. Yaakub and M. F. J Klaib, "A Random Search Based Effective Algorithm for Pairwise Test Data Generation" in proc. of IEEE International Conference on Electrical Control and Computer Engineering 2011, Kuantan, Malaysia, 2011,pp 293 - 297.

[10] X. Qu and M. B. Cohen. " A study in prioritization for higher strength combinatorial testing". The 2nd International Workshop on Combinatorial Testing, 2013.

[11] T. Nanba, T. Tsuchiya, and T. Kikuno. "Using satisfiability solving for pairwise testing in the presence of constraints". IEICE Transactions, vol 95(9), 2012, pp.1501–1505.

[12] D. K. R. Chaudhuri and T. Zhu, "A Recursive Method for Construction of Designs," Discrete Mathematics - Elsevier, vol. 106, 1992, pp. 399-406.

[13] M. F. J. Klaib, K. Z. Zamli, N. A. M. Isa, M. I. Younis, and R. Abdullah, "G2Way – A Backtracking Strategy for Pairwise Test Data Generation," in Proc. of the 15th IEEE Asia-Pacific Software Engineering Conf., Beijing, China, 2008, pp. 463-470.

[14] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester. "Test suite prioritization by cost-based combinatorial interaction coverage". International Journal of Systems Assurance Engineering and Management, vol 2(2) , 2011, pp.126–134.

[15] J. Petke, S. Yoo, M. B. Cohen, M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing", in Proc.ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 26-36, USA, 2013.

[16] S. Varshney, M. Mehrotra. Search based software test data generation for structural testing: a perspective, in ACM SIGSOFT Software Engineering Notes archive, vol 38 (4), NY, USA , 2013, pp. 1-6.

[17] M. B. Cohen, J. Snyder, and G. Rothermel, "Testing Across Configurations: Implications for Combinatorial Testing," in Proc. of the 2nd Workshop on Advances in Model Based Software Testing, Raleigh, North Carolina, USA, 2006, pp. 1-9.

[18] C. J. Colbourn, M. B. Cohen, and R. C. Turban, "A Deterministic Density Algorithm for Pairwise Interaction Coverage," in Proc. of the IASTED Intl. Conference on Software Engineerin, Innsbruck, Austria, 2004, pp. 345-352.

[19] K. C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," IEEE Transactions on Software Engineering, vol. 28, 2002, pp. 109-111.

[20] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing," in Proc. of the 28th Annual Intl. Computer Software and Applications Conf. (COMPSAC'04), Hong Kong, 2004, pp. 72-77.

[21] Mats Grindal, "Handling Combinatorial Explosion in Software Testing", Linkoping Studies in Science and Technology, Dissertation No. 1073, Sweden, 2007

[22] K. Z. Zamli, N. A. M. Isa, M. F. J. Klaib, Z. H. C. Soh and C. Z. Zulkifli, "On Combinatorial Explosion Problem for Software Configuration Testing," in Proc. of the International Robotics, Vision, Information and Signal Processing Conference (ROVISP2007), Penang, Malaysia, 2007.

[23] R. Kuhn, R. Kacker, Y. Lei, "Combinatorial Software Testing," IEEE Transactions on Software Technologies, August 2009, pp. 94-96.

[24] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton, "The Automatic Efficient Test Generator (AETG) System," in Proc. of the 5th International Symposium on Software Reliability Engineering, Monterey, CA, USA, 1994, pp. 303-309.

[25] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing," in Proc. of the 3rd IEEE Intl. High-Assurance Systems Engineering Symp., Washington, DC, USA, 1998, pp. 254-261.

[26] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOD: Efficient Test Generation for Multi-Way Software Testing," Journal of Software Testing, Verification, and Reliability, vol. 18, 2009, pp. 125-148.

[27] J. Bach, "Allpairs Test Case Generation Tool," Available from: http://tejasconsulting.com/open-testware/feature/allpairs.html

[28] "TConfig," Available from: http://www.site.uottawa.ca/~awilliam/.

[29] "Jenny," Available from: http://www.burtleburtle.net/bob/math/.

[30] "TVG," Available from: http://sourceforge.net/projects/tvg.

[31] M. B. Cohen, "Designing Test Suites for Software Interaction Testing," PhD in Computer Science. New Zealand: University of Auckland, 2004.

[32] M. Grindal, J. Offutt, and S. F. Andler, "Combination Testing Strategies: a Survey," Software Testing Verification and Reliability, vol. 15, 2005, pp. 167-200.

[33] M. Grindal, B. Lindstrom, J. Offutty, S. F. Andler, "An Evaluation of Combination Strategies for Test Case Selection", Technical Report HS-IDA-TR-03-001, Department of Computer Science, University of Skövde, 2003.

[34] D.R. Kuhn, R.N. Kacker and Y. Lei, Combinatorial Coverage as an Aspect of Test Quality, the Journal of Defense Software Engineering,2014.

[35] D.R. Kuhn, R.N. Kacker and Y. Lei, Measuring and Specifying Combinatorial Coverage of Test Input Configurations, Innovations in Systems and Software Engineering: a NASA journal, 2014.

[36] J. Torres-Jimenez, I. Izquierdo-Marquez, Survey of Covering Arrays, 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2013), Timisoara, Romania, 23-26, 2013, pp. 20-27.

[37] R.N. Kacker, D.R. Kuhn, Y. Lei, and J.F. Lawrence, Combinatorial Testing for Software: an Adaptation of Design of Experiments,Measurement, vol. 46, no. 9, 2013, pp. 3745-3752.

[38] X. Niu, C. Nie, Y. Lei, A.T.S. Chan, Identifying Failure-Inducing Combinations Using Tuple Relationships, 2nd International Workshop on Combinatorial Testing (IWCT 2013), in Proceedings of the Sixth IEEE International Conference on Software, Testing, Verification and Validation (ICST 2013), Luxembourg, March 18-22, 2013, pp. 271-280.

[39] M.N. Borazjany, L.S.G. Ghandehari, Y. Lei, R.N. Kacker and D.R. Kuhn,An Input Space Modeling Methodology for Combinatorial Testing, 2nd International Workshop on Combinatorial Testing (IWCT 2013), in Proceedings of the Sixth IEEE International Conference on Software, Testing, Verification and Validation (ICST 2013), Luxembourg, March 18-22, 2013, pp. 372-381.