Journal of Computer Science 6 (5): 542-547, 2010
ISSN 1549-3636
© 2010 Science Publications

Tree Based Test Case Generation and Cost Calculation Strategy for Uniform Parametric Pairwise Testing

Mohammad F.J. Klaib, Sangeetha Muthuraman, Noraziah Ahmad and Roslina Sidek
School of Computer Systems and Software Engineering,
University Malaysia Pahang, 26300 Gambang, Pahang, Malaysia

Abstract: Problem statement: Although it is very important to test any system extensively it is usually too expensive to do so owing to the cost and the resources that are involved in it. Software testing is a very important phase of software development to ensure that the developed system is reliable. Some systematic approach for testing is essential to test any system and make it acceptable. Combinatorial software interaction testing is one which tests all possible software interactions. This interaction could be at various levels such as two way interaction (pairwise) or three or four or five or six way interactions. Combinatorial interaction testing had been used in several fields. It was reported in literature that pairwise combinatorial interaction testing had identified most of the software faults. **Approach:** In this study we proposed a new strategy for test suite generation, a tree generation strategy for pairwise combinatorial software testing, with parameters of equal values. The algorithm considered one parameter at a time systematically to generate the tree until all the parameters were considered. This strategy used a cost calculation technique iteratively for each of the leaf nodes to generate the test suite until all the combinations were covered. **Results:** The experimental data showed that we had achieved about 88% (or more in some cases) of reduction in the number of test cases needed for a complete pairwise combinatorial software interaction testing. **Conclusion:** Thus, the strategy proposed had achieved a significant reduction in minimizing the number of test cases that was generated.

Key words: Combinatorial testing, software testing, pairwise testing

INTRODUCTION

Software testing is a very important phase of the software development cycle (Bryce *et al*., 2005; Tsui and Karam, 2007). A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of coverage; a test set achieves 100% coverage if it completely satisfies the criterion. Coverage is measured in terms of the requirements that are imposed. Partial coverage is defined to be the percent of requirements that are satisfied. Test requirements are specific things that must be satisfied or covered. Example: In case of ‘for statement’ coverage, each statement within the ‘for’ is a requirement. In mutation, each mutant is a requirement. In combinatorial testing the covering array is a requirement.

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its requirements. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation.

What is Combinatorial Explosion? Combinatorial Explosion (Grindal, 2007; Zamli *et al*., 2007a; 2007b) describes the effect of functions that grow very rapidly as a result of combinatorial considerations. Consider for instance testing the addition functionality of simple calculator. Restricting the input space to only positive integers still yields a large number of possible test cases, (1+1; 1+2; 1+3; :::; 1+N; 2+1; 2+2; :::;N+N), where N is the largest integer that the calculator can represent. The example stated above highlights the combinatorial explosion problem.

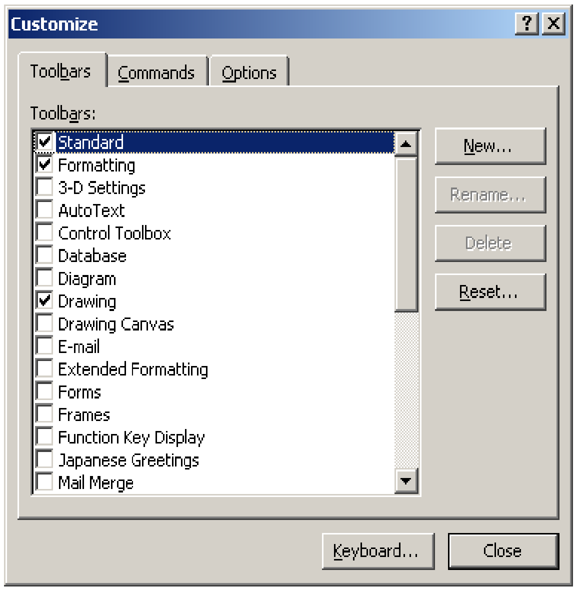
To be more clear on how the problem of combinatorial explosion could be resource and time consuming, consider for instance testing the customize dialog in the tools menu of Microsoft Word as shown in

Corresponding Author: Mohammad F.J. Klaib, School of Computer Systems and Software Engineering, University Malaysia Pahang, 26300 Gambang, Pahang, Malaysia

Fig. 1. Even if only the toolbar tab is considered, there are 31 checkboxes to be tested. Therefore there are 2^{31} (i.e., 2147483648) combinations of test cases to be evaluated. If the time required for one test case to be evaluated is 5 min, then it would require nearly 20428 years for a complete test of the toolbar tab alone! Therefore, it is very clear that combinatorial explosion is a serious issue which has to be considered and software testing always faces the problem of combinatorial explosion.

Although it is important to test any software exhaustively, it is not practically possible to do so in reality owing to the cost and resources (Chaudhuri and Zhu, 1992; Klaib *et al*., 2008; Copeland, 2004) that are needed for the tests to be conducted. Therefore, one good solution is to construct a test suite with an acceptable number of test cases for any t-way testing (Burr and Young, 1998; Cohen *et al*., 1997; 2008; Zamli *et al*., 2007c; Lei *et al*., 2009). There have been some solutions already proposed (Cohen *et al*., 1994; Cohen, 2004; Lei and Tai, 1998; Shiba *et al*., 2004), however the problem of constructing the minimum test set for t-way testing is NP-complete (Shiba *et al*., 2004; Tai and Lei, 2002) and the challenges in this field still remain.

Pairwise testing (Dalal *et al*., 1999; Kuhn and Reilly, 2002; Kuhn and Okum, 2006; Kuhn *et al*., 2004; Yan and Zhang, 2008; Bryce and Colbourn, 2006) is an approach whereby every combination of valid values of all the parameters should be covered by at least one test case. Combinatorial pairwise approaches (Grindal *et al*., 2005) to testing are used in several fields and have recently gained momentum in the field of software testing through software interaction testing. Pairwise testing provides a systematic approach to identify and isolate faults, since many faults are caused by unexpected 2-way interactions among system factors. Empirical results show that 50-97% of the software faults could be identified by pairwise interaction testing (Klaib *et al*., 2008; Cohen *et al*., 1997; Lei and Tai, 1998; Dalal *et al*., 1999; Kuhn *et al*., 2008). This study proposes an efficient tree generation and cost calculation strategy for constructing a test suite with minimum number of test cases.



Toolbars:

- ✓ Standard
- ✓ Formatting
- ☐ 3-D Settings
- ☐ AutoText
- ☐ Control Toolbox
- ☐ Database
- ☐ Diagram
- ✓ Drawing
- ☐ Drawing Canvas
- ☐ E-mail
- ☐ Extended Formatting
- ☐ Forms
- ☐ Frames
- ☐ Function Key Display
- ☐ Japanese Greetings
- ☐ Mail Merge

Fig. 1: Customize tab of the Microsoft word software

MATERIALS AND METHODS

The proposed strategy: This strategy proposed constructs the tree based on the parameters and the values given to it. It considers one parameter at a time to construct the tree until all the values of all the parameters are considered. To illustrate the concept consider a system with parameters and values as shown below:

- Parameter A has values A1 and A2
- Parameter B has values B1 and B2
- Parameter C has values C1 and C2

The algorithm first uses all the values of the first parameter to construct the tree. Then it uses all the values of the second parameter and then the third. 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. Figure 2 shows how the tree would be constructed.

Once the tree construction is over we have all the test cases generated and the cost calculation can begin. The cost calculation algorithm calculates the cost of each of the leaf nodes or test cases. The cost of any leaf node or test case is equal to the number of pairs that it covers in the covering array. 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 starts the calculation of the cost of each and every leaf node in order. Once it reaches a leaf node with the maximum cost it includes this node or test case into the test suite and also deletes all the pairs that the test case has covered in the covering array.

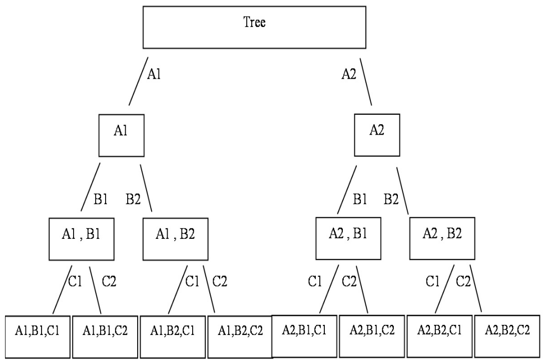


Fig. 2: Test tree

Table 1: Test generation process

Test cases	Costs	Test suite
A1,B1,C1	3	T1
A1,B1,C2	2	-
A1,B2,C1	2	-
A1,B2,C2	3	T2
A2,B1,C1	2	-
A2,B1,C2	3	T3
A2,B2,C1	3	T4
A2,B2,C2	0	-

Thus in the first iteration all the test cases included in the test suite are said to have the maximum cost. If all the pairs in the covering array are covered then the algorithm stops else it goes to the second iteration. Now the maximum cost value (Wmax) is decreased by one and the next best test cases i.e. test cases that can cover the next maximum number of pairs are chosen and included in the test suite and the corresponding pairs covered by these test cases deleted from the covering array. Thus the algorithm continues until all the pairs are covered. For the example in Fig. 2 all the test cases which are included in the test suite are identified in a single iteration as shown in Table 1. There are four test cases included in the test suite that covers all the pairwise interactions.

Table 1 explains how the cost calculation is done iteratively for Figure 2. This example is an exception where all the test cases needed for pairwise interaction have been included in a single iteration. However, it takes more iteration for other samples. The strategies work very efficiently in identifying the minimum number of test cases for any given parameters with uniform values for pairwise combinatorial testing.

The tree generation strategy for test case generation:

Strategy tree generation:

```
Begin
    {for the first parameter p1 }
        T = {(v_1), (v_2)...... (vj) / v_1, v_2 and vj are values of
        p1 and are sequentially connected}
        If n=1 then stop;

        {For the remaining parameters}

        For parameter p_i, i =2, 3 ....n do
        Begin

        For each Test (v_1, v_2, ......, vi-1) 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

            Replace the original with v_1 and all the replicated
            tests with (v_2, v_3.......v_{i-1}, v_i etc. respectively)
            Where v_i is a value of p_i and each of which is
            considered in order

            End
        End
        End
End
```

The tree generation strategy thus provides the following advantages:

- A systematic method whereby all possible test cases are generated in order
- The above procedure works fine with the parameters having any number of values. Therefore all parameters can have different or same values as any real time system to be tested would have
- The procedure appears to generate the full tree by using all the values of the parameters but at every iteration only a set of leaf nodes are left thus having a list of leaf nodes (or test cases) when the procedure 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 of the implementation we have constructed the same tree as in Fig. 2 using the proposed tree generation algorithm. This proposed algorithm constructs the tree by minimizing the number of nodes. Minimization of the number of nodes is achieved by giving importance only to the leaf nodes at every stage.

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. The existing set of leaf nodes be Esoln, new set of leaf nodes be Nsoln and the number of values of the parameter under consideration be n. Then:

$$\text{Nsoln} = \text{Esoln} * n$$

Let there be 4 leaf nodes and the next parameter to be considered has 2 values. Then the new list of nodes will have 8 new leaf nodes as a result. The algorithm considers every leaf node separately and calculates the number of times this particular node needs to be replicated with the formulae given below:

$$\text{The number of values of } p_i\text{-}1$$

where, p_i- is the i^{th} parameter under consideration for constructing the new set of leaf nodes and i = 1,2,….N- the number of parameters. In the Fig. 2 that is shown above consider the leaf nodes (A1, B1), (A1, B2), (A2, B1) and (A2, B2). To construct the next level of leaf nodes the parameter under consideration is C, which has values C1 and C2. Therefore, the node (A1, B1) needs to be replicated once. Now we will have two (A1, B1) nodes to which C1 is added to the first and C2 is added to the second and then the replicated node is included in the list of leaf nodes after the original node and before the node (A1, B2). The same is done to (A1, B2). It is replicated once and hence we have two of it (one original and one replicated node). Now C1 is added to the first (original node) and C2 is added to the second (replicated node). Thus we have (A1, B2, C1) and (A1, B2, C2). The same process is done for the nodes (A2, B1) and (A2, B2) and as a result we get (A2, B1, C1), (A2, B1, C2) and (A2, B2, C1), (A2,B2, C2) respectively. If there are more parameters the same is continued until all the parameters are considered. Thus, once the list of leaf nodes is generated we go to the next strategy of iterative cost calculation to construct the test suite.

Test suite generation by iterative cost calculation strategy: Strategy test suite generation by iterative cost calculation:

```
Begin

Generate the pairwise covering array for the given parameters.

Create a cost array corresponding to the T list.

Initialize each element in the cost array to infinity (highest value).

Let T' be an empty set.

Wmax = N(N-1)/2. // 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 the pairs that Tj covers in the covering array

   Cost[Tj] = The number of pairs 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 the pairs marked in the covering array
End

   Wmax--;
End
End
```

Table 2: Covering array

A with B	A with C	B with C
A1,B1	A1,C1	B1,C1
A1,B2	A1,C2	B1,C2
A2,B1	A2,C1	B2,C1
A2,B2	A2,C2	B2,C2

Table 3: Experimental results

System	Exhaustive No. of test cases	TBGCC	Reduction (%)
S1	8	4	50.00
S2	27	10	62.96
S3	81	9	88.88
S4	32	6	81.25
S5	64	17	73.40
S6	16	6	62.50

The above algorithm starts by constructing the covering array as shown in Table 2. Table 2 shows the covering array for the example shown in Fig. 2. In the second step it creates and initializes a cost array corresponding to the T List. Then the algorithm iterates through the list of test cases T to generate the test suite T' until all the pairs of the covering array are covered. At each iteration all the test cases with the maximum cost (Wmax) for that particular iteration are included in the test suite. Thus the algorithm guarantees identifying a minimum set of test cases for parameters with same number of values.

RESULTS

We have implemented a Tree Based Test Case Generation and Cost Calculation Tool called TBGCC that includes the above strategies. We have presented the result for six system configurations as shown in Table 3. Thus, the strategies proposed works well in constructing a minimum number of test cases which covers all pairwise interactions. Table 3 shows the exhaustive number of test cases and the percentage of reduction achieved. We observe that as the number of parameters and its values increases there is a significant reduction in the number of test cases included into the test suite.

The six system configurations used are as follows:

S1: 3 2-valued parameters
S2: 3 3-valued parameters
S3: 4 3-valued parameters
S4: 5 2-valued parameters
S5: 3 4-valued parameters
S6: 4 2-valued parameters

DISCUSSION

In this study we have proposed the tree based test case generation and iterative cost calculation strategy for pairwise testing. Both the strategies proposed have been implemented. Both the algorithms presented have worked well for 2-way testing with uniform parametric values. However the algorithms could be extended for non uniform values. These algorithms could also be extended further and used for higher t-way interaction testing.

CONCLUSION

Both the strategies implemented in this study works well for uniform parametric values. The tree generation strategy works well in generating the test tree. The iterative cost calculation strategy works well in achieving a good amount of reduction in the test size (in some cases more than 88%). Therefore, the above strategies proposed have generated an efficient number of test cases that covers all combinatorial pairwise interactions for uniform parametric values.

REFERENCES

Bryce, R. and C.J. Colbourn, 2006. Prioritized interaction testing for pairwise coverage with seeding and avoids. Inform. Software Technol. J., 48: 960-970.

Bryce, R., C.J. Colbourn and M.B. Cohen, 2005. A framework of greedy methods for constructing interaction tests. Proceeding of the 27th International Conference on Software Engineering, May 2005, ACM Press, St. Louis, MO., USA., pp: 146-155. DOI: 10.1145/1062455.1062495

Burr, K. and W. Young, 1998. Combinatorial test techniques: Table-based automation, test generation and code coverage. Proceeding of the International Conference on Software Testing, Analysis and Review, Oct. 1998, San Diego, CA., pp: 503-513.

Chaudhuri, D.K.R. and T. Zhu, 1992. A recursive method for construction of designs. Discrete Math., 106: 399-406.

Cohen, D.M., S.R. Dalal, A. Kajla and G.C. Patton, 1994. The Automatic Efficient Test Generator (AETG) system. Proceeding of the 5th International Symposium on Software Reliability Engineering, Monterey, (SREM'94), CA., USA., pp: 303-309. http://aetgweb.argreenhouse.com/papers/1994-issre

Cohen, D.M., S.R. Dalal, M.L. Fredman and G.C. Patton, 1997. The AETG system: An approach to testing based on combinatorial design. IEEE Trans. Software Eng., 23: 437-444. DOI: 10.1109/32.605761

Cohen, M.B., 2004. Designing test suites for software interaction testing. Degree of Doctor of Philosophy Thesis, Department of Computer Science, The University of Auckland. http://cse.unl.edu/~myra/papers/mbcdiss.pdf

Cohen, M.B., C.J. Colbourn and A.C.H. Ling, 2008. Constructing strength three covering arrays with augmented annealing. Discrete Math., 308: 2709-2722.

Copeland, L., 2004. A Practitioner's Guide to Software Test Design. Artech House, Boston, MA., ISBN: 9781580537919, pp: 294.

Dalal, S.R., A. Jain, N. Karunanithi, J.M. Leaton and C.M. Lott *et al*., 1999. Model based testing in practice. Proceeding of the International Conference on Software Engineering, May 1999, pp: 285-294. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.4894&rep

Grindal, M., J. Offutt and S.F. Andler, 2005. Combination testing strategies: A survey. Software Test. Verific. Reliab., 15: 167-200.

Grindal, M., 2007. Handling combinatorial explosion in software testing. Ph.D. Thesis, Dissertation No. 1073, Linkoping Studies in Science and Technology, University of Skövde and Enea, Sweden. http://www.artes.uu.se/publications/Grindal.html

Klaib, M.F.J., K.Z. Zamli, N.A.M. Isa, M.I. Younis and R. Abdullah, 2008. G2Way-a backtracking strategy for pairwise test data generation. Proceeding of the 15th IEEE Conference on Asia-Pacific Software Engineering, Dec. 3-5, IEEE Xplore Press, Beijing, China, pp: 463-470. DOI: 10.1109/APSEC.2008.49

Kuhn, D.R. and M.J. Reilly, 2002. An investigation of the applicability of design of experiments to software testing. Proceeding of the 27th NASA/IEEE Software Engineering Workshop, Dec. 5-6, IEEE Computer Society, Washington DC., USA., pp: 69-80. DOI: 10.1109/SEW.2002.1199454

Kuhn, D.R. and V. Okum, 2006. Pseudo-exhaustive testing for software. Proceeding of the 30th Annual IEEE/NASA Software Engineering Workshop, Apr. 24-28, IEEE Computer Society, Washington DC., USA., 2006, pp: 153-158. http://portal.acm.org/citation.cfm?id=1264143

Kuhn, D.R., D.R. Wallace and A.M. Gallo, 2004. Software fault interactions and implications for software testing. IEEE Trans. Software Eng., 30: 418-421.

Kuhn, D.R., Y. Lei and R. Kacker, 2008. Practical combinatorial testing: beyond pairwise. IT Profession. J., 10: 19-23. http://csrc.nist.gov/groups/SNS/acts/itpro-final.pdf

Lei, Y. and K.C. Tai, 1998. In-parameter-order: A test generation strategy for pairwise testing. Proceeding of the 3rd IEEE International Symposium on High-Assurance Systems Engineering, June 21-21, Washington DC., USA., pp: 254-261. DOI: 10.1109/HASE.1998.731623

Lei, Y., R. Kacker, D. Kuhn, V. Okun and J. Lawrence, 2009. IPOG/IPOD: Efficient test generation for multi-way software testing. J. Software Test. Verific. Reliab., 18: 125-148. DOI: 10.1002/stvr

Shiba, T., T. Tsuchiya and T. Kikuno, 2004. Using artificial life techniques to generate test cases for combinatorial testing. Proceeding of the 28th Annual International Conference on Computer Software and Applications, Sept. 28-30, IEEE Computer Society, Washington DC., USA., pp: 72-77. http://portal.acm.org/citation.cfm?id=1025478

Tai, K.C. and Y. Lei, 2002. A test generation strategy for pairwise testing. IEEE Trans. Software Eng., 28: 2004, 109-111.

Tsui, F.F. and O. Karam, 2007. Essentials of Software Engineering. 2nd Edn., Jones and Bartlett Publishers, Massachusetts, USA., ISBN: 9780763785345, pp: 416.

Yan, J. and J. Zhang, 2008. A backtracking search tool for constructing combinatorial test suites. J. Syst. Software, 81: 1681-1693.

Zamli, K.Z., M.F.J. Klaib and N.A.M. Isa, 2007a. Combinatorial explosion problem in software testing: Issues and practical remedies. Proceeding of the 3rd Malaysian Software Engineering Conference-Striving for High Quality Software, (SHQS'07), Selangor, Malaysia, pp: 24-28.

Zamli, K.Z., N.A.M. Isa, M.F.J. Klaib, Z.H.C. Soh and C.Z. Zulkifli, 2007b. On combinatorial explosion problem for software configuration testing. Proceeding of the International Conference on Robotics, Vision, Information and Signal Processing, July 20-22, Penang, Malaysia, pp: 442-446.

Zamli, K.Z., N.A.M. Isa, M.F.J. Klaib and S. Norbaya, 2007c. A tool for automated test data generation (and execution) based on combinatorial approach. Int. J. Software Eng. Appli., 1: 19-34.