Sensors & Transducers, Vol. 160, Issue 12, December 2013, pp. 130-136

Sensors & Transducers

© 2013 by IFSA
http://www.sensorsportal.com

Generating Combinatorial Test Suite with Solution Space Tree for Configurations Testing of Sensors Networks

Ziyuan WANG, Qin YUAN
School of computer, Nanjing University of Posts and Telecommunications,
Xinmofan Road 66, Gulou District, Nanjing, 210003, China
Tel.: +86-25-85866422, fax: +86-25-85866433
E-mail: wangziyuan@njupt.edu.cn

Received: 25 September 2013 /Accepted: 22 November 2013 /Published: 30 December 2013

Abstract: There are many results about generating pair-wise covering arrays with strength τ=2 have been reported, but fewer results are published for high-strength covering arrays with a higher-strength τ>2. In configuration testing of sensor networks, high-strength covering array is required to construct combinatorial test cases. To generate combinatorial test suite with higher-strength, a backtracking algorithms, which is based on solution space tree, is proposed in this paper by extending an existing pair-wise combinatorial test suite generation algorithm. In solution space tree model, each test case is represented as a path from the root to a leaf node in the tree. And proposed algorithm generates test cases one by one, by backtracking depth-first searching in the solution space tree. Finally, to assess the efficiency of proposed algorithm, computational comparison with other published methods is reported. *Copyright © 2013 IFSA.*

Keywords: Combinatorial testing, Test generation, Solution space tree, Configuration testing, Censor network.

1. Introduction

Sensor networks have been widely used in real life. A complex sensor network may be affected by many parameters or factors, such as configurations, internal events, external inputs etc, where each parameter may have numerous available options. For a sensor network with complex configuration space, a simple and intuitive configuration testing approach is named as Each Choice, which requires each option of each configuration parameter to be included in at least one test case [10]. However, it is reported from numerous case studies that many faults may be involved in the combinations of options from different parameters [8]. This phenomenon shows us that, rather than the single parameter or factor, the interaction of multiple configuration parameters may also affect the work of sensor networks.

When testing interactions in configuration space, combinatorial testing is considered as a practical technique, since it uses a small test suite that cover all needed parametric values and their combinations, to detect the faults triggered by these single parameters and even the interactions of them. E.g., for a configuration space that has n parameters, it may be unacceptable to cover all the n-tuple combinations of parametric options or values, since the combinatorial explosion. Meanwhile, combinatorial testing could provide a tradeoff between the cost of testing and the degree of combinatorial interaction coverage. E.g., pair-wise combinatorial testing requires covering all the 2-tuple combinations of parametric values rather than the n-tuples.

For instance, considering the configuration space of an imaginary sensor network, in which there are totally 4 configuration parameters, including sensor

nodes, connection, operation system, and database. Each configuration parameter has 3 available options (see Table 1). Exhaustive testing, which covers all the possible 4-tuple combinations, requires 3^4=81 test cases. Meanwhile, the pair-wise combinatorial testing requires only 9 test cases to cover all possible pairs of parametric values, which are shown Table 2.

Table 1. Configuration space.

Nodes	OS	Connection	DB
Air Sensor Speed Sensor Pressure Sensor	Linux Windows Macintosh	LAN ISDN Modem	DB/2 Oracle Access

Table 2. Pair-wise combinatorial test suite.

No.	Nodes	OS	Connection	DB
1	Air Sensor	Linux	LAN	DB/2
2	Air Sensor	Windows	ISDN	Oracle
3	Air Sensor	Macintosh	Modem	Access
4	Speed Sensor	Linux	ISDN	Access
5	Speed Sensor	Windows	LAN	Oracle
6	Speed Sensor	Macintosh	Modem	DB/2
7	Pressure Sensor	Linux	Modem	Oracle
8	Pressure Sensor	Windows	LAN	Access
9	Pressure Sensor	Macintosh	ISDN	DB/2

Because of the power of combinatorial testing, many results about generating combinatorial test suites have been reported in past years. Most of them focus on the generation of pair-wise combinatorial test suites (or combinatorial test suites with strength 2), but fewer results are published for high-strength combinatorial test suites. It means that the problem of generating high-strength combinatorial test suite is still not well solved today. Therefore, in order to generate high-strength combinatorial test suite, a backtracking algorithms, which is based on solution space tree, is proposed in this paper by extending an existing pair-wise combinatorial test suite generation algorithm. In solution space tree model, each test case is represented as a path from the root to a leaf node in the tree. And proposed algorithm generates test cases one by one, by backtracking depth-first searching in the solution space tree. We have implemented this algorithm as a tool, and the result shows that it has some good properties and merits, and it can be a complement of the existed methods and tools.

The remainder of this paper is organized as follows: section 2 describes the definitions about combinatorial testing. In section 3, we describe the model of solution space tree, propose backtracking algorithm for generating high-strength combinatorial test suite, and discuss some properties. Section 4 reviews related works. Section 5 compares proposed algorithm with some existed algorithms and tools. Finally, conclusion remarks are given in section 6.

2. Definitions

Combinatorial test suite is designed based on the covering array, which is special kind of mathematical structure. Considering a configuration space that has n configuration parameters $c_1, c_2,\dots, c_n$. We suppose each configuration parameter c_i has a_i discrete values f_i has a_i parametric values (i=1, 2, …, n). Without loss of generality, we use $C=\{c_1, c_2,\dots, c_n\}$ to denote the set of these n parameters, and $T_i=\{1, 2,\dots, a_i\}$ to denote the set of valid values for the parameter c_i (i=1, 2, …, n). Here we can define $a=\max_{1\le i\le n}\{a_i\}$. If the cardinalities of all these value sets are equivalent ($a=a_1=a_2=\dots=a_n$), we say it is a configuration space with fixed-level factors, and these parameters could be also denoted as $C=\{a^n\}$. Otherwise, we say it is a configuration space with mixed-level factors.

Definition 1. A n-tuple $(v_1, v_2,\dots, v_n)$ $(v_1\in T_1, v_2\in T_2,\dots, v_n\in T_n)$ is a **test case** or **test data** for the given configuration space.

Definition 2. Let t_1 and t_2 are test cases for a configuration space, if there are b same values in the b same positions of the two n-tuples, we call that the **overlap degree** of the two test data is b.

For example, (2,1,3,2,1,3) and (3,1,2,2,1,1) are two test cases and their overlap degree is 3. If the overlap degree of the two test cases has the property that $b\geq 2$, there are at least one value combination of corresponding b parameters is covered by the both two test cases. None of the combinations is covered by both the test data when $b\leq 1$.

For the fixed-level systems, we have orthogonal array and fixed-level covering array.

Definition 3. A τ-way **orthogonal array** $OA_\lambda(m; \tau, n, a)$ is an $m\times n$ array on totally a symbols with the property that each $m\times\tau$ sub-array contains all ordered subsets from a symbols of size τ exactly λ times.

Definition 4. A τ-way fixed-level **covering array** (or called fixed-level covering array with strength τ) $CA(m; \tau, n, a)$ is an $m\times n$ array on totally a symbols with the property that each $m\times\tau$ sub-array contains all ordered subsets from a symbols of size τ at least once.

Similarly, for the fixed-level systems, we have mixed-level covering array.

Definition 5. A τ-way mixed-level **covering array** (or called mixed-level covering array with strength τ) $CA\ (m; \tau, (a_1, a_2, \dots, a_n))$ is an $m\times n$ array on totally a symbols, the j-th column contains only the elements from the set T_j of size a_j ($1\leq j\leq n$), and each $m\times\tau$ sub-array contains all τ-tuple combinations of values from the τ columns at least once.

Definition 6. A τ-way fixed-level or mixed-level covering array is a **smallest τ-way fixed-level or mixed-level covering array**, if the number of rows is as small as possible.

Both τ-way fixed-level covering array and τ-way mixed-level covering array could be called as τ-way covering array. A τ-way combinatorial test suite could be obtained easily from a τ-way covering array,

where each row of covering array is a test case of combinatorial test suite. Therefore, we assume that the terms "combinatorial test suite" and "covering array" are equivalent in this paper.

We can conclude from above definitions that an orthogonal array with λ=1 is a kind of fixed-level covering array, the reverse is not right. The size of mixed-level covering array is usually smaller than the size of orthogonal array with λ>1 [2]. E.g., for 100 parameters with two values each, a 2-way orthogonal array requires 128 tests with λ=32, while 10 test cases are sufficient to cover all pairs in a 2-way covering array. In software testing or configuration testing, it is only necessary to cover the combinations of parameter values once and not necessary to cover them with the same number of times. So covering array, which can improve the efficiency and decrease the cost of testing with the smaller test suite, is much more practical than orthogonal array.

3. Algorithm

We used to proposed a backtracking algorithm, which is based on solution space tree, for generating pair-wise combinatorial test suite previously [9]. In the following section, we will extend that algorithm for high-strength combinatorial test generation, and improve the process of backtracking search for the efficiency of test generation.

3.1. Solution Space Tree Model

For a configuration space which has n parameters and each parameter c_i has a_i values where we can let $a_1 \geq a_2 \geq \ldots \geq a_n$ without loss of generality, each test case can be represented as a path from the root to a leaf node in the tree. All the usable test case forms a tree as follows: The root of the tree has a_1 child branches which represent a_1 values of parameter c_1 respectively; each root in the second level of the tree has a_2 child branches which represent a_2 values of parameter c_2 respectively; …; each root in the n-th level of the tree has a_n child branches which represent a_n values of parameter c_n respectively. For example, when n=3 and a_1=a_2=a_3=3, all the usable test case forms a solution space tree as Fig. 1.

Combinatorial test suite generation is to find out a subset of paths from the solution space tree. For example, the 9 paths: 1-1-1, 1-2-2, 1-3-3, 2-1-2, 2-2-3, 2-3-1, 3-1-3, 3-2-1 and 3-3-2 form a test suite with 9 test cases: {(1,1,1), (1,2,2), (1,3,3), (2,1,2), (2,2,3), (2,3,1), (3,1,3), (3,2,1), (3,3,2)}, which satisfies the requirement of pair-wise covering array and is the smallest pair-wise covering array.

3.2. Backtracking Algorithm to Generate High-strength Combinatorial Test Cases

The backtracking algorithm for generating high-strength combinatorial test cases in solution space tree consists of four steps:

Step 1: Assign some seed test cases into a test set TS. The seed test case could be: the ones that are taken care of by testers; the ones that are assumed to be failure-trigger test cases; the ones that generated by other testing techniques; etc. If people don't assign any seed test cases, then TS=$\varnothing$.

Step 2: Backtracking search using the depth-first strategy in the solution space tree to select test cases one by one until the searching process is end. The selected test cases, whose overlap degree with all the existing test cases in the set TS is no more than τ-1, will be put into TS. The detail procedure can be found in Algorithm 1 and 2. At the beginning of test generation, procedure BackTrack (1) should be called.

Step 3: Check whether all the test cases in the set TS could cover all the required τ-tuple combinations of parametric values in configuration space. If it is true, then algorithm ends; else, we list all the τ-tuple combinations of parametric values that have not been covered by test cases in TS. It will take running time of O(n^2) to do this step.

Step 4: Construct test cases with one-test-at-a-time algorithm to cover left τ-tuple combinations. The process of one-test-at-a-time strategy is described in Algorithm 3. The concrete algorithms based on that strategy include AETG [3], Density Algorithm [11], TCG [12], etc. All these concrete algorithm could be adopted in this step.

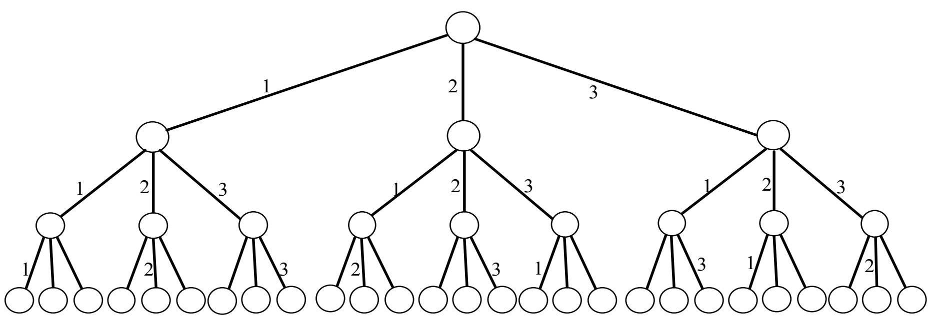

Fig. 1. The solution space tree when n=3,a_1=a_2=a_3=3.

Algorithm 1. Backtracking Searching on Solution Space Tree.

```
Procedure BackTrack(i)
Begin
    If (i = n+1) then
        /* values of all n factors have been fixed */
        Add current_test into test suite
    Else
        Insert all a_i values (1, 2, …, a_i) of factor f_i
        into node_vector as a given order
        For j = 1 to a_i
            current_test[j]= node_vector[j]
            /* IsFit check whether the overlap
            between current test and others is less
            than the given upper bound τ-1*/
            If (IsFit(current_test, i)) then
                BackTrack(i+1)
            End If
        End For
    End If
End
```

Algorithm 2. Check Overlap Degree

```
Procedure IsFit(current_test, deep)
Begin
    For each test in test suite
        overlap=0
        For i = 1 to deep
            If (current_test[i]=test[i]) then
                overlap++
            End If
            If (overlap > upper_bound) then
                Return false
            End If
        End For
    End For
    Return True
End
```

Algorithm 3. One-test-at-a-time Strategy

```
Input: CombSet: a set of required τ-tuple
    combinations
Output: a test suite
Begin
    UncovCombSet:=CombSet
    While (UncovCombSet≠∅)
        Generate one test case current_test to cover
        τ-tuples combinations in UncovCombSet as
        more as possible
        Add the current_test into test suite, modify
        UncovCombSet by removing τ-tuples that
        covered by current_test
    End While
End
```

We analyze the time complexity of proposed algorithm. The number of candidate paths in solution space tree is $\prod_{1\leq i\leq n} a_i < a^n$ When check overlap degree for a given path, at most n nodes should be checked. And for each node, values of correspond parameters in at most a^τ test cases should be compared. Therefore, the worst time complexity of generating a τ-way combinatorial test suite is $O(n \times a^{n+\tau})$. In practical, the time performance is much better than the worst value, since the number of nodes that should be checked is much smaller.

3.3. An Improved Searching Algorithm

The time performance of backtracking algorithm could be improved again. It can be concluded that, after the procedure BackTrack(n) is called and a test case is selected, the BackTrack(n-1) will be called to search in another path. But the BackTrack(n-1) can not find any test cases, because the overlap between current path and the last selected test case is n-1>τ-1. So after finding a test case, we should call BackTrack(τ) instead of BackTrack(n-1). The improved algorithm named BackTrackE is described in Algorithm 4.

Algorithm 4. Improved Backtracking Searching on Solution Space Tree

```
flag = False
Procedure BackTrackE(i)
Begin
    If (i = n+1) then
        /* values of all n factors have been fixed */
        Add current_test into test suite
        flag = True
    Else
        Insert all a_i values (1, 2, …, a_i) of factor f_i
        into node_vector as a given order
        For j = 1 to a_i
            /* if a test has been selected, the deep
            must fall back to 1+upper_bound */
            If ((flag = True) then
                If (i>1+upper_bound)) then
                    Break
                Else
                    flag = False
                End If
            End If
            current_test[j]= node_vector[j]
            /* IsFit check whether the overlap
            between current test and others is less
            than the given upper bound τ-1 */
            If (IsFit(current_test, i)) then
                BackTrack(i+1)
            End If
        End For
    End If
End
```

It is clear that, the worst time complexity of the procedure BackTrackE is equal to that of procedure BackTrack. But in actual, the former is efficient than the latter, since it check less nodes in solution space tree.

3.4. Properties of Algorithm

In the algorithm model there is a one-to-one correspondence between all the paths in the solution space tree and all the usable test data for SUT. The test data generation for pair-wise testing is to search a subset of paths from the solution space tree. The algorithm has the following properties:

Proposition 1 The heuristic algorithm base on the solution space tree model can generate test suite for pair-wise testing on the basis of an assigned test data set by the testers. Such that the generated test suite not only satisfy the intention of testers, but also satisfy the requirement of pair-wise testing.

Proposition 2 For a configuration space, assume TS= $\varnothing$, if $a_1=a_2=\ldots=a_n=a=p$ (or p^m), $n\leq p+1$ (or p^m+1), where p is a prime, then the test suit generated by Algorithm 1 and 4 is orthogonal array $OA_1(a^\tau;\tau, n, a)$ with $\lambda=1$.

Proof: By the construction theory of orthogonal array [1], when $a_1=a_2=\ldots=a_n=a$ ($a=p$ or $a=p^m$), $n\leq p+1(p^m+1)$, where p is a prime, there exists an orthogonal array. It is a $a^\tau\times n$ matrix and every row of it is a test data, so the overlap degree between any two of the test data in covering array is no more than τ-1, otherwise, there must exists two test data, the overlap degree is equal to τ or more, then the two test data cover a τ-tuple combination twice. Since there are a^τ distinct τ-tuple combination between any τ parameters, there should be also at least a^τ test data to cover them. So there must exist a pair uncovered by the test suite generated from orthogonal array.

Corresponding to the solution space tree, there exist r^2 paths that their overlap degree to each other is no more than 1. Algorithm 1 is used to search all the paths with the overlap degree of no more than 1 to each other. So the test suite generated by algorithm 1 is an orthogonal array $OA_1(a^\tau;\tau, n, a)$.

4. Related Works

Since we like to minimize the testing cost as much as possible, we are interested in generating the least test suite for pair-wise testing, known as the smallest pair-wise covering array. However, the problem of finding the smallest pair-wise covering array is NP-complete. There are two main pragmatic approaches towards the problem. One is the algebraic approach. Various algebraic have been proposed for finding the smallest pair-wise covering array.

The original approach is to use orthogonal arrays [1], but orthogonal arrays have a balance requirement that every pair is covered the same number of times, and this requirement make it impractical for software testing. A. W. Williams presented a construction method based on some basic blocks, and developed a new, fast, deterministic algorithm for achieving pair-wise interaction coverage [6]. Noritaka Kobayashi et al. also propose a new algebraic construction and give an upper bound on the size of test set generated. The results show that the proposed construction can generate very small 2-factor covering designs [5]. Although these algebraic constructions are very effective when all parameters have the same number of values, they cannot well deal with the case where parameters have different numbers of values.

Another approach is to use the heuristics method. D. M. Cohen et al. proposed a heuristic search-based approach, which has been implemented as a test generation system, called AETG [3]. TCG [12] and DDA [11] are similar to the AETG. K. C. Tai and Y. Lei proposed a new test generation strategy, called in-parameter-order (or IPO), for pair-wise testing, and they have also implemented it as a tool, called PairTest [4]. Generally test sets generated by these approaches tend to be larger than those generated by the algebraic methods, and they cannot guarantee bounds on the size of resulting test sets.

And besides the algebraic methods and heuristics methods, another types of methods are meta-heuristic algorithms, including generic algorithm, simulated annealing, ant colony algorithm, etc [14]. These algorithms could generate small combinatorial test suites, but require massive execution time.

5. Experimental Results

To assess the efficiency of proposed algorithms, we compare them to some existed algorithms and tools. In experiment, we compare proposed algorithm to some other algorithms and tools, including DDA [11], TCG [12], GREEDY [7], TVG [16], PICT [13], AETG [3], GA [14], ACA [14], GA-N [15], IPO [4], and Jenny [17]. Note that our proposed algorithms BackTrack and BackTrackE will output the same covering array, so we only illustrate one result for each input (see “SST” in Table 3).

As displayed in such a table, we find out ACA and GA generates the smallest test suite for almost all inputs. The reason is that ACA and GA are both meta-heuristic algorithms. By ignoring the data about ACA and GA, both SST and GREEDY generate the smallest test suites for 4 of all 8 inputs. Therefore, it could be concluded from experimental result that, the test suites generated by our proposed algorithms are much smaller than that generated by most heuristic algorithms, hough their performances are worse than that of ACA and GA that belongs to meta-heuristic.

Table 3. Sizes of generated 3-way combinatorial test suites.

	SST	DDA	TCG	GREEDY	TVG	PICT	AETG	GA	ACA	GA-N	IPO	Jenny
S_1	37	47	53	43	48	48	38	33	33	52	48	51
S_2	64	64	106	64	120	111	77	64	64	85	64	112
S_3	125	211	225	184	239	215	194	125	125	223	200	215
S_4	332	359	363	325	409	369	330	331	330	389	366	373
S_5	1462	1587	1624	1474	1949	1622	1473	1501	1496	1769	1678	1572
S_6	223	237	225	220	269	241	218	218	218	336	239	236
S_7	109	116	108	106	133	119	114	108	106	120	120	130
S_8	363	369	377	388	429	368	377	360	361	373	464	397

(S_1: 3^6; S_2: 4^5; S_3: 5^6; S_4: 6^6; S_5: 10^6; S_6: 5^7; S_7: $5^2 4^2 3^2$; S_8: $10^1 6^2 4^3 3^1$)

6. Conclusions

Exhaustive testing is impractical and impossible since the combinatorial explosion of configuration space of high-configuration systems, e.g. sensor networks. So it is a key issue to select the minimal test suite for the effective configuration testing. The reduced test suite with good quality can improve the efficiency and decrease the cost. In this paper, we proposed two extending algorithm, which is based on the model of solution space tree, to generating high-strength combinatorial test suite for configuration testing. These algorithms can generate good covering array as the approximation of the least high-strength covering array. The better approximation still needs the further research on the better algorithm.

There have been many results on combinatorial testing, but some problems are still necessary to be studied in the future. For example, new test generation techniques are required. Corresponding techniques for test prioritization, value constraint, fault location, and regression testing are also very necessary to be studied.

Acknowledgements

The works described in this paper were supported by the National NSF of China (61003020, 61300054); NSF of Jiangsu Province (BK2011190, BK20130879); NSF for College & University in Jiangsu Province (13KJB520018); Foundation of NJUPT (NY212023); Open Foundation of Guangxi Key Lab of Trustworthy Software (KX201328).

References

[1]. A. S. Hedayat et al., Orthogonal Arrays: Theory and Applications, *Springer-Verlag*, USA, 1999.

[2]. I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino. Applying design of experiments to software testing: experience report, in *Proceedings of the 19th International Conference on Software Engineering*, Boston, Massachusetts, United States, Vol. 5, 1997, 205-215.

[3]. D. M. Cohen et al., The AETG System: An Approach to Testing Based on Combinatorial Design, *IEEE Transactions on Software Engineering*, Vol. 23, Issue 7, July 1997, pp. 437-444.

[4]. Y. Lei, K. C. Tai, In_Parameter_Oder: A Test Generation Strategy for Pair-wise Testing, Technical Report TR-2001-03. Dept. of Computer Science, *North Carolina State Univ.,* Raleigh, North Carolina, Mar. 2001.

[5]. N. Kobayashi, T. Tsuchiya, T. Kikuno, A New Method for Constructing Pair-wise Covering Designs for Software Testing, *Information Processing Letters*, Vol. 81, Issue 2, 2002, pp. 85-91.

[6]. A. W. Williams, Software component interaction testing: coverage measurement and generation of configurations, PhD Thesis, Ottawa-Carleton Institute for Computer Science, School of Information Technology and Engineering, *University of Ottawa,* Canada, 2002.

[7]. P. J. Schroeder, B. Korel, Black-box test reduction using input-output analysis, in *Proceedings of the ISSTA'00,* Portland, Oregon, 2000, pp. 173-177.

[8]. D. R. Kuhn, A. M. Gallo, Software Fault Interactions and Implications for Software Testing, *IEEE Transaction on Software Engineering*, Vol. 30, Issue 6, June 2004, pp. 1-4.

[9]. N. Changhai, X. Baowen, S. Liang, W. Ziyuan, A new heuristic for test suite generation for pair-wise testing, in *Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006.

[10]. M. Grindal, B. Lindstrom, J. Offutt, S. F. Andler, An Evaluation of Combination Strategies for Test Case Selection, *Empirical Software Engineering*, Vol. 11, 2006, pp. 583-611.

[11]. R. C. Bryce, C. J. Colbourn, A Density-Based Greedy Algorithm for Higher Strength Covering Arrays, *Software Testing, Verification and Reliability*, Vol. 19, Issue 1, 2009, pp. 37-53.

[12]. Y. Tung, W. S. Aldiwan, Automating Test Case Generation for the New Generation Mission Software System, in *Proceedings of the IEEE Aerospace Conference,* Big Sky, Montana, USA, 2000, pp. 431-437.

[13]. J. Czerwonka, Pairwise Testing in Real World: Practical Extensions to Test Case Generator, in *Proceedings of the 24th Pacific Northwest Software*

Quality Conference, October 9-11, 2006, pp. 419-430.
[14]. T. Shiba, T. Tsuchiya, T. Kikuno, Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing, in *Proceedings of 28th Annual International Computer Software and Applications Conference (COMPSAC' 04)*, Hong Kong, China, Vol. 1, September, 2004, pp. 72-78.
[15]. C. Nie, B. Xu, L. Shi, G. Dong, Automatic Test Generation for N-way Combinatorial Testing, *Quality of Software Architectures and Software Quality, Lecture Notes in Computer Science,* Vol. 3712, 2005, pp. 203-211.
[16]. http://sourceforge.net/projects/tvg/
[17]. http://burtleburtle.net/bob/math/jenny.html

___

2013 Copyright ©, International Frequency Sensor Association (IFSA). All rights reserved.
(http://www.sensorsportal.com)