VARIABLE STRENGTH T-WAY TEST SUITE GENERATOR WITH CONSTRAINTS SUPPORT

Rozmie R. Othman[1],Norazlina Khamis[2],Kamal Z.Zamli[3*]

[1] School of Computer and Communication, Universiti Malaysia Perlis (UniMAP), PO Box 77, d/a Pejabat Pos Besar, 01007 Kangar, Perlis, Malaysia

[2] Faculty of Computing & Informatics
Universiti Malaysia of Sabah,
88400 Kota Kinabalu, Sabah

[3]Faculty of Computer Systems and Software Engineering, Universiti Malaysia Pahang, Lebuhraya Tun Razak, 26300 Kuantan, Pahang, Malaysia

*Corresponding Author. Email: kamalz@ump.edu.my

ABSTRACT
T-way testing (or interaction testing) is a common test planning method used to sample a complete or exhaustive test suite systematically.In t-way testing, it is assumed that interaction only occurs between t numbers of parameters (where t is the interaction strength). Therefore, all t-way strategies generate the t-way test suite with the intention to cover every possible combination produces by the interacting parameters (or also known as tuples). However, for some systems under test(SUT), there are some combinations that are known to produce invalid outputs or eventrigger unwanted errors. Additionally, there are also some known combinations that are impossible to occur due to requirements set to the system. As such, these combinations (termed constraints) have to be excluded from the final test suite. Whilemany t-way strategies have been proposed in literature for the past 20 years (e.g. GTWay, MIPOG, TConfig and TCG), only IPOG and PICT strategies have been known to support constraints in variable strength test suite generation. However, as t-way test suite generation process is an NP-hard problem, no single strategy can claim dominance over the others. Motivated by the challenges, this paper proposes a new strategy named General Variable Strength with Constraints (GVS_CONST) that support variable strength interaction with constraints consideration. Empirical evidence demonstrates that in most cases GVS_CONST outperforms other competing strategies in term of test suite size.

Keywords: Interaction Testing, Variable Strength Interaction, T-Way Testing, Constraints, Combinatorial Testing.

1.0 INTRODUCTION

Every input for any piece of software is impractical to test exhaustively[1-3]. Consider an integer type input (i.e. a 32-bit variable) which can hold more than 4 billion possible values, testing all possible values will require a very long time and seems impossible for any software development activity. As a result, techniques like equivalent partitioning and boundary analysis have been proposed to help test engineers to convert a very large number of possible input values into a much smaller set without losing any fault detection capability[4]. However, as system under test (SUT) can be influenced by many inputs, exhaustive testing again seems to be impractically especially for a SUT which consist of many inputs.

As such, many sampling techniques (e.g. random testing, each-choice and base-choice, anti-random and t-way testing) have been proposed for the past 20 years to help test engineers select only a subset of test cases (i.e. from the exhaustive pool of test cases) that would produce an acceptable level of confidence in fault detection capability [5-8]. Amongst the proposed sampling techniques, t-way testing is the most common sampling technique used to sample exhaustive test suite systematically. Compare to the others, t-way testing offers a fair distribution of sampling mechanism (as compared to random testing) and giving test engineers the flexibility of choosing the interaction strength to suite the SUT (unlike each-choice which forces test engineers into a 1-way testing). In addition, t-way testing able to reduce the number of test cases significantly and at the same time able to achieve maximum fault detection capability.

In t-way testing, it is assumed that interaction only occurs between t numbers of parameters (where t is the interaction strength). Therefore, all t-way strategies generate the t-way test suite with the intention to cover every possible combination produces by the interacting parameters (or also known as tuples). However, for some SUT, there are combinations that are known to produce invalid outputs or even trigger unwanted errors. Additionally, there are also some known combinations that are impossible to occur due to requirements set to the system. As such, these combinations (termed constraints) have to be excluded from the final test suite. While many t-way strategies have been proposed in literature for the past 20 years (e.g. GTWay[5, 6], MIPOG[7-10], TConfig [11, 12]and TCG[13]), few strategies have sufficiently considered constraints during test generation process. mAETG[14], SA[14] and TestCover[15] are among the t-way strategies that support constraints in t-way test suite generation. Although these strategies address the constraints issue successfully, the implementation is limited to uniform strength interaction only (i.e. all interactions consist a fixed number of parameters (t)). In many real applications, interactions between input parameters are hardly uniform. Recently, many variable strength strategies have been proposed in literature (e.g. ITTDG [16], VS-PSTG [17, 18], IPOG [19, 20], ACS [21], PICT [22, 23] and Density [24]), however only IPOG and PICT strategies are known to support constraints in variable strength test suite generation. In addition, as test suite generation process in an NP-hard problem [25, 26], no single strategy can claim dominance over the others. Motivated by the challenges, this paper proposes a variable strength t-way test suite generator strategy with constraints support named General Variable Strength with Constraints (GVS_CONST). Empirical evidence demonstrates that in most cases GVS_CONST outperforms other competing strategies in term of test suite size.

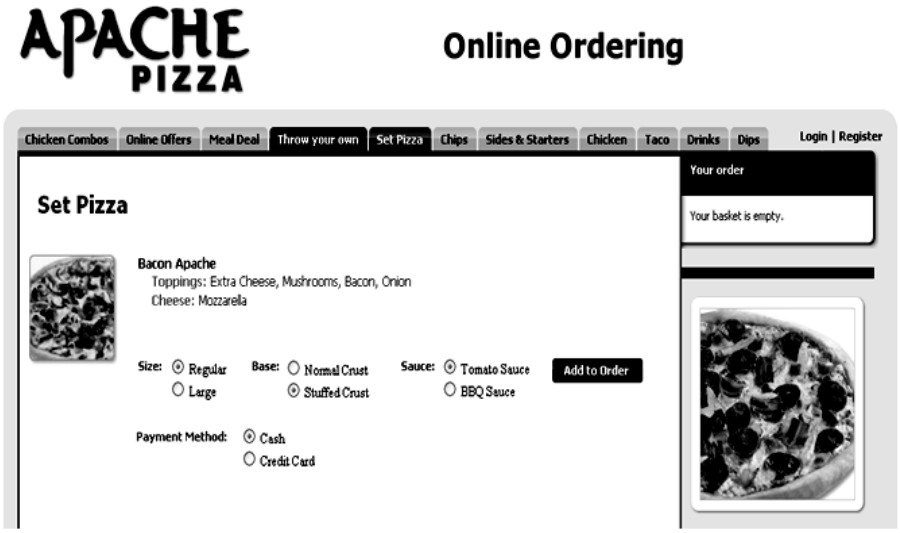
The rest of this paper is organized as follows. Section 2 gives the background on t-way testing and constraints while section 3 presents our related works. In section 4, we present our propose strategy, GVS_CONST and in section 5 we evaluate the performance of GVS_CONST against other competing strategies. Finally section 6 gives our conclusion.

2.0 BACKGROUND ON VARIABLE STRENGTH T-WAY TEST SUITE WITH CONSTRAINTS

T-way testing samples test cases from exhaustive test suite based on parameter interaction. By sampling the exhaustive test suite, the generated t-way test suite will be much smaller and therefore reduce the time as well as cost required for test execution. To further illustrate the idea of t-way testing, consider an online pizza ordering system shown in Fig. 1. The system consists of 4 input parameters (i.e. size, base, sauce and payment method) and every parameter consists of 2 values shown in Fig. 1 as well. In order to aid the discussion, all parameters and values will be represented using symbolic values depicted in Table 1.

For this online pizza ordering system, the exhaustive test suite (shown in Table 2) can be generated using parameter-value combination shown in Table 1. It should be noted that, exhaustive test suite is actually a full strength t-way test suite since it is assumed that all parameters are interacting (or in this case can be referred as a 4-way test suite since the interaction strength (t) is 4). Here, the exhaustive test suite consists of 16 test cases (i.e. $2^4 = 16$).

Let's consider a uniform 2-way testing. In 2-way testing, it is assumed that every possible combination of 2 parameters (e.g. AB, AC, AD, BC, BD and CD) is interacting. Here, the final test suite should covers all parameter-value combinations (or tuples) produce by every interacting parameter. Table 3 shows every possible 2-way tuple and one possible exhaustive test case that can cover the tuple.



Input Parameters	Size	Base	Sauce	Payment Method
Values	Regular	Normal Crust	Tomato Sauce	Cash
	Large	Stuffed Crust	BBQ Sauce	Credit Card

Fig. 1.Online Pizza Ordering System

Table 1.Symbolic Value Representation

Input Parameters	A	B	C	D
Values	a1	b1	c1	d1
	a2	b2	c2	d2

Table 2.Exhaustive Test Suite

Test Case No. T#	A	B	C	D
1	a1	b1	c1	d1
2	a1	b1	c1	d2
3	a1	b1	c2	d1
4	a1	b1	c2	d2
5	a1	b2	c1	d1
6	a1	b2	c1	d2
7	a1	b2	c2	d1
8	a1	b2	c2	d2
9	a2	b1	c1	d1
10	a2	b1	c1	d2
11	a2	b1	c2	d1
12	a2	b1	c2	d2
13	a2	b2	c1	d1
14	a2	b2	c1	d2
15	a2	b2	c2	d1
16	a2	b2	c2	d2

Table 3.2-way Tuples and Possible Exhaustive Test Case That Can Cover the Tuple

Interacting Parameter	Tuples Produce	T#	Interacting Parameter	Tuples Produce	T#
A,B	a1,b1	1	A,C	a1,c1	1
	a1,b2	5		a1,c2	3
	a2,b1	9		a2,c1	9
	a2,b2	13		a2,c2	11
A,D	a1,d1	1	B,C	b1,c1	1
	a1,d2	2		b1,c2	3
	a2,d1	9		b2,c1	5
	a2,d2	10		b2,c2	7
B,D	b1,d1	1	C,D	c1,d1	1
	b1,d2	2		c1,d2	2
	b2,d1	5		c2,d1	3
	b2,d2	8		c2,d2	4

From Table 3, it can be seen that not all exhaustive test cases are required to cover every 2-way tuples. Thus, removing unwanted exhaustive test cases, 2-way test suite can be formed as shown in Table 4. Compared to exhaustive test suite, 2-way test suite consists of only 11 test cases (a 38% reduction from exhaustive test suite). Using the covering array notation from [26] and [27], the uniform strength interaction test suite, *F*, can be expressed as in Equation 1.

$$F = \mathrm{CA}(N, t, C) \quad (1)$$

where, N = the number of test data inside the final test suite.
t = the interaction strength
C = value configuration can be represented as following: ${v_1}^{p_1}, {v_2}^{p_2}, \ldots, {v_n}^{p_n}$ which indicate that there are p_1 parameters with v_1 values, p_2 parameters with v_2 values and so on.

Table 4.2-way Test Suite, CA($11, 2, 2^4$)

T# From Exhaustive Test Suite	2-Way Test Cases
1	a1, b1, c1, d1
2	a1, b1, c1, d2
3	a1, b1, c2, d1
4	a1, b1, c2, d2
5	a1, b2, c1, d1
7	a1, b2, c2, d1
8	a1, b2, c2, d2
9	a2, b1, c1, d1
10	a2, b1, c1, d2
11	a2, b1, c2, d1
13	a2, b2, c1, d1

The concept of parameters interaction in uniform strength t-way testing applies in variable strength t-way testing as well. The difference between variable strength and uniform strength t-way testing is on the number of interaction strength assigned for a particular system. In uniform strength t-way testing, single interaction strength is assumed for all parameters while in variable strength t-way testing, instead of having uniform interaction strength for all parameters, any subset of parameters can have different interaction strength. Using the same 2-way testing example from above and assuming that parameter ABC is interacting (3-way testing for parameter A, B and C), a variable strength test suite can be generated using the same method as shown in Table

3 with extra contribution from interacting parameter ABC (shown in Table 5 and 6). Here 12 test cases are needed for variable strength interaction, a reduction of 25% from exhaustive test suite. Similar to uniform strength test suite, variable strength test suite can be expressed using covering array notation represented by Equation 2.

$$F = \text{VCA}(N, t, C, S) \quad (2)$$

where, N = the number of test data inside the final test suite.

t = the interaction strength

C = value configuration can be represented as following: ${v_1}^{p_1}, {v_2}^{p_2}, \ldots, {v_n}^{p_n}$ which indicate that there are p_1 parameters with v_1 values, p_2 parameters with v_2 values and so on.

S = multiple subset of interaction strength represented using Equation 1.

Table 5.Extra Tuples Contribute by Interacting Parameter ABC

Tuples Produce	T#	Tuples Produce	T#
a1,b1,c1	1	a2,b1,c1	9
a1,b1,c2	3	a2,b1,c2	11
a1,b2,c1	5	a2,b2,c1	13
a1,b2,c2	7	a2,b2,c2	15

Table 6.Variable Strength Test Suite, VCA($12, 2, 2^4$, CA($3,2^3$))

T# From Exhaustive Test Suite	2-Way Test Cases
1	a1, b1, c1, d1
2	a1, b1, c1, d2
3	a1, b1, c2, d1
4	a1, b1, c2, d2
5	a1, b2, c1, d1
7	a1, b2, c2, d1
8	a1, b2, c2, d2
9	a2, b1, c1, d1
10	a2, b1, c1, d2
11	a2, b1, c2, d1
13	a2, b2, c1, d1
15	a2, b2, c2, d1

While both variable strength and uniform strength t-way testing assume that all parameters are interacting, there might be a special case where certain combination of parameter-values might lead to fatal error or logically the combination cannot be executed. Let's say the above online pizza ordering system has following requirements:-

1. Credit Card can only be used for purchasing the large pizza.
2. BBQ sauce only available for stuffed crust pizza.

With these special requirements, 2 constraints can be derived. First, any purchase using credit card (i.e. d2) for regular size pizza (i.e. a1) is invalid. Second, combination between BBQ sauce (i.e. c2) and normal crust pizza (i.e. b1) is also invalid. Revisiting Table 4 with these constraints, it can be noted that T# 2, 4, 8 and 11 cannot be executed as the case cases are against the system requirements. This proved the importance of constraints consideration during test suite generation. Table 7 shows other generated 2-way test suite that satisfies the system requirements. Here, every test case consists of credit card, the pizza size is large while test case consists

of BBQ sauce the pizza base is stuffed crust. In order to incorporate constraints information in covering array notation, the following term is added to existing covering array notation:

$Ct(\{c_{1,1},c_{4,Ct}, \{c_{2,1},c_{3,2}, \ldots\} \ldots)$ where $c_{a,b}$ refers to b^{th} value of a^{th} parameter. (3)

Table 7.2-way Test Suite with Constraints, CA (5, 2,24) $Ct(\{c_{1,1},c_{4,2}\},\{c_{2,1},c_{3,2}\})$

T# From Exhaustive Test Suite	2-Way Test Cases with Constraints
1	a1, b1, c1, d1
7	a1, b2, c2, d1
10	a2, b1, c1, d2
13	a2, b2, c1, d1
16	a2, b2, c2, d2

3.0 RELATED WORK

Many t-way strategies have been proposed by researchers for the past 20 years. In the early year, most of the proposed t-way strategies focus on uniform t-way interaction. Earlier work stresses on (uniform) pairwise strategies (where t=2). Here, most early existing pairwise strategies are based on orthogonal array (such as (OA) [28, 29], MOA [30]). Although having fast execution time, the applicability of orthogonal array is often restricted to small configurations [31, 32].

Owing to the aforementioned limitations, the focus gradually shifted to uniform t-way strategies. Strategies like AETG [33], Jenny [34], TConfig [11], GTWay [6], TCG [13] and MIPOG [7] are among strategy that support uniform strength interaction. Later, researchers found that interaction between input parameters are hardly uniform. As such several strategies such as ITTDG [16], VS-PSTG [17, 18], IPOG [19, 20], ACS [21], PICT [22, 23] and Density [24] have been proposed that support variable strength interaction. As this paper focuses on variable strength interaction, uniform strength t-way strategies have been purposely ignored.Interested readers are referred to survey work byGrindal in [35].

ITTDG is one of t-way strategy that supports variable strength interaction. The strategy can be characterized as one-test-at-a-time strategy since one test case is generated iteratively in order to cover all interaction tuples. For each test case, ITTDG strategy generates several test case candidates and selects the best candidates (one that covered the most uncovered tuples) as final test case. Like AETG, ITTDG adopts iterative and random heuristics for test case selection. Unlike AETG which generates new test case candidates for every iterations; ITTDG only generates new test case candidates when there is a tie situation (i.e. when more than one value can cover the most uncovered tuples).

PICT [23] is a test suite generation strategy created by Czerwonka and has been widely used in Microsoft for software testing purposes. Initially, PICT generates all possible tuples bases on the SUT's configuration and marks every tuple as an uncovered tuple. If there are any constraints declare by test engineer, the tuples involve in those constraints will be mark as excluded. After that, one uncovered tuple will be selected and extended until completion using greedy heuristic method (i.e. cover as many uncovered tuples as possible but without any excluded tuples). The completed test case will be put in final test suite and all the tuples covered by this test case will be mark as covered. This process will be repeated until there is no tuples mark as uncovered.

Density is another one-test-at-a-time strategy similar to ITTDG and PICT. Density generates test case based on mathematical formula which derived from density properties. The used of density concept in generating t-way test suite is first introduced by Bryce in [36, 37] and been used in generating uniform t-way test suite. Later, Wang et al [24] extends the density concept by introducing the formulae for local and global density in order to support the variable strength interaction. Both local and global density formulae can be obtain from [24, 38].

Apart from the computational approach as highlighted earlier, there are also several attempts of generating t-way test suite using artificial intelligence (AI) approach. VS-PSTG and ACS are two t-way strategies that utilize particle swarm optimization and ant colony optimization respectively. For VS-PSTG, the searching for best test

case is inspired by the behavior of flocks of birds. Internally, the strategy iteratively combines local and global search in order to find the best test cases to greedily cover the given interaction tuples. As for ACS, the test case is searched by colonies of ants on some possible paths. The paths qualities are evaluated in terms of the pheromones which signify convergence. Here, the optimum path corresponds to the best test case to be included in the final test suite. Similar to ITTDG, PICT and Density, both VS-PSTG and ACS are one-test-at-a-time strategy. On the other hand, IPOG is an example of one-parameter-at-a-time strategy. IPOG starts the generation process by generating an exhaustive test suite for the first t parameters (in the case of variable interaction strength (t), the highest t will be chosen) as the initial test suite. Later, IPOG relies on two processes called horizontal extension and vertical extension. Horizontal extension is a process of adding one parameter to the initial test suite. This process is repeated until all parameters are covered by the test suite. Vertical extension is a complementary process for horizontal extension in order to ensure that all tuples are covered. During horizontal extension, there is a possibility that several tuples cannot be covered by the initial test suite. In this case, the initial test suite will be extended vertically by adding several new test cases to the initial test suite.

4.0 GVS_CONST IMPLEMENTATION

4.1 GVS_CONST Strategy

GVS_CONST strategy is inspired by earlier work in GTWay [5, 6]. Here, GVS_CONST implements the merger rule uses in GTWay with some modifications to support for constraints. GVS_CONST differs from GTWay on the rule to decide which test case to be select into the final test suite. In GTWay, a complete test case is generated using merger rule and will be added into final test suite only if the generated test case covers the most uncovered tuples. However in GVS_CONST, the maximum coverage is checked and maintained during the merging process so that every complete test case will always been added into final test suite. As a result, a significant time reduction for test suite generation can be achieved. The modified merger rule is as follows:-

1. Two tuples can only be merged when they are combinable (i.e., each tuple parameter complements the other tuple missing parameter or both tuple share the common parameter-value combination).
2. Two tuples will be merged only if the result from the merging process achieves maximum tuples coverage and adheres to the constraints.
3. In cases when some tuples cannot be merged (due to values that are not uniform), the value for other parameters will be selected randomly as long as the selected values are consistent with the constraints.

```
Input: System configuration, SC and list of constraints, C
Output: Final Test Suite, T

Begin:
Generate list of tuple based on SC and store in Ct

While (Ct is not empty)
  P = get first tuple from Ct
  Q = get other tuple from Ct
  while P is not a complete test case
    if P and Q can be merge and agree with all constraints in C
      merge P and Q to form new P
    end if
    Q = get other tuple from Ct
  end while
  store P in T
  removed tuples covered by P from Ct
End
```

Fig. 2.GVS_CONSTAlgorithm

To further illustrate GVS_CONST, refer to the system elaborated in Section 2. In this case, it is assumed that the system requires a 2-way testing with similar constraints (i.e. {a1,d2} and {b1,c2}). Initially, GVS_CONST strategy generates a list of possible tuples (i.e. similar to what have been shown in Table 3) and removes all tuples correspond to the constraints. In order to reduce tuples generation time as well as tuple uncovered checking process (as required by second merger rule), GVS_CONST adopts a data structure named tuple tree. Details on the implementation of tuple tree will be explained in the next section. Once tuple tree has been initialized, GVS_CONST strategy start to traverse the list and select first found tuple (i.e. {a1,b1}) in order to begin the merging process. After selecting one tuple, the strategy starts to find other tuple to be merged with selected tuple following the merger rules. Here, tuple {c1,d1} will be selected, as the tuple satisfies the merger rules (i.e. merging these tuples produce a complete test case {a1,b1,c1,d1} that covers the most uncovered tuples and consistent with the constraints). The result of the merging process (i.e. the complete test case) will push into final test suite and all tuples covered by the test case will be removed from the list. It should be noted that, here one merging process is required to form a complete test case. There is possibility where several merging processes are required in order to form a complete test case. Then, the whole process will be repeated until all uncovered tuples are covered by the final test suite. Overall, the GVS_CONST strategy can be summarized by Fig. 2.

4.2 Tuple Tree

To choose the right type of data structure to hold covered tuples, one important issue is the time required to access the covered tuples (i.e. owing to the fact that most of the generation time is spent on checking whether or not the tuples are covered). As number of tuples can easily reach hundreds of thousands, an improper selection of data structure may slow down the generation process significantly.

To address the aforementioned timing issues, GVS_CONST implements a modified tree structure, called tuple tree, as a data structure to store all the covered tuples. Like other tree structures, tuple tree uses node (represented by the circle) and branch (represented by arrow) to store the information regarding covered tuples. Nodes represent the values of parameter and connected nodes (i.e. via branches) form a tuple. For illustration, consider the running example shown in Section 2 with interaction strength 2. Fig. 3 depicts the tuple tree when four tuples (i.e. a0,b0,X,X, a0,X,c1,X , a0,X,X,d0 and X,X,c0,d1) are stored.

Concerning the access time (i.e. time requires to check whether a particular tuple is stored in a tuple tree or not), it can be clearly seen from Fig. 3 that the number of required iterations are equal to number of parameters (in this case is four). Here, the numbers of iterations are far less than some conventional data structures (e.g. Array List and Hash Set) which may require iteration of all the tuples in order to decide whether or not a particular tuple has already been added to the list.

Apart from the access time issue, another equally important issue relates to the amount of memory required to store the tuples. The memory requirement for a tuple tree depends on the number of nodes inside the tree. The larger the numbers of tuples are, the larger the memory requirement. In order to address this issue, terminal node (represented by double circle), is used to substitute several nodes that have been covered.

Terminal node refers to a node without any branches. Here, terminal node can be used to imply that all required tuples has already been covered and inserted into the tuple tree (without the need to traverse to all parameters or levels). For illustration, consider tuple tree shown in Fig. 4 where terminal node have been found in parameter b level. Here, the terminal node in the tuple tree simply means that any tuples start with a0,X (i.e. a0,X,c0,X , a0,X,c1,X , a0,X,X,d0 and a0,X,X,d1) have been added into this tuple tree.

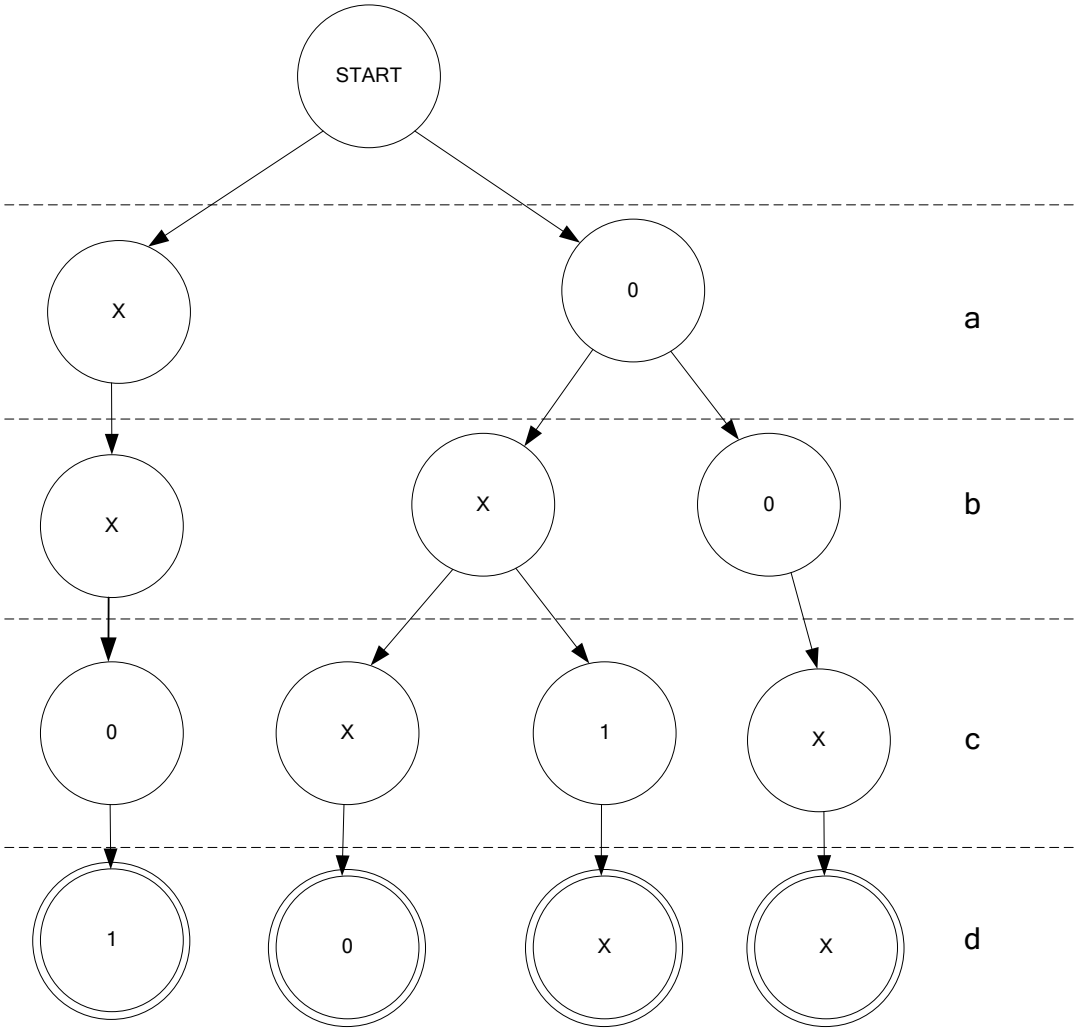


Fig. 3. Example of Tuple Tree

As mentioned earlier, terminal node combines several nodes that have been completed. Therefore, when adding a new tuple, every node related to that tuple will be checked for completeness in order to be converted to terminal node. Here, a node can be converted to a terminal node only when the node has k+1 number of branches (where k is the number of value for the next parameter and 1 to represent don't care value) and all branches are pointed to terminal node. Here, every node located at the lowest level of tuple tree (i.e. the last parameter) can always be represented as terminal node.

In general, there are two advantages when using tuple tree with terminal node. Firstly, the memory required to store covered tuples can be reduced by replacing nodes that consist maximum number of branches (i.e. all values for the next parameter are already covered) with a single terminal node. Secondly, the access time also will potentially be reduced as now there is no need to traverse to all levels (or parameters).

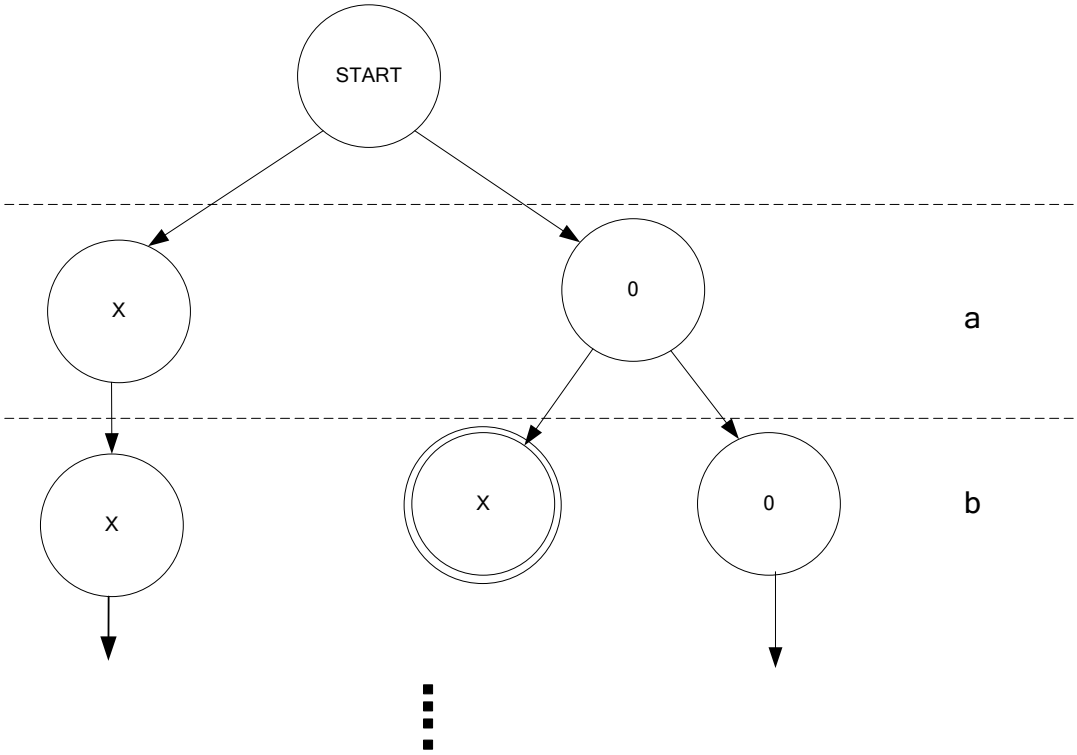


Fig. 4. Example of Terminal Node in Tuple Tree

5.0 EMPIRICAL EVALUATION OF THE PROPOSED STRATEGY

Our goal is to evaluate the performance of GVS_CONST against other competing strategies (i.e. GTWay, IPOG and PICT) in term of generated test suite size. 7 system configurations have been used in this evaluation and the test suite size generated by each strategy is depicted in Table 8. It should be noted that bold value represents the optimal test suite size. "NS" symbol (which means not supported) is used in all constraints test suite for GTWay since GTWay merger rule does not allowed any constraints insertion.

From Table 8, we can see that GVS_CONST produces the optimal test suite size for most system configurations except for VCA(N, 3,47,CA(4,45)) without constraints and VCA(N, 4,49,CA(5,46)) where IPOG produces the optimal test suite size. In addition, GVS_CONST produces smaller test suite size compared to GTWay in all system configuration. Overall, PICT always produces the worst test suite size. On a positive note, PICT tend to produce smaller constraints test suite as compared to unconstraint test suite while GVS_CONST and IPOG do not necessary to produce smaller constraints test suite. This is due to the greedy nature of PICT strategy that generates test case to cover the most uncovered tuples. Since the number of tuples need to be covered are reduced by the constraints, the generated test suite also will be reduced.

Table 8.Generated Test Suite Size

System Configuration	Constraints Info.	GVS_CONST	GTWAY	IPOG	PICT
		N			
VCA(N,3,3^7,CA(4,3^5))	Ct()	**118**	125	119	900
	Ct({$c_{1,2}$,$c_{4,1}$,$c_{7,1}$},{$c_{2,1}$, $c_{6,2}$,$c_{7,2}$},{$c_{3,1}$,$c_{4,1}$,$c_{5,3}$}, {$c_{2,3}$,$c_{4,2}$,$c_{6,3}$}, {$c_{2,2}$,$c_{4,1}$,$c_{7,3}$}, {$c_{1,1}$,$c_{2,2}$,$c_{5,3}$,$c_{6,2}$})	**112**	NS	116	765
VCA(N, 3,4^7,CA(4,4^5))	Ct()	365	377	**352**	5328
	Ct({$c_{3,2}$,$c_{5,3}$}, {$c_{2,2}$,$c_{6,4}$,$c_{7,4}$}, {$c_{1,3}$,$c_{5,3}$,$c_{6,2}$,$c_{7,1}$}, {$c_{5,2}$,$c_{6,2}$,$c_{7,3}$}, {$c_{1,4}$,$c_{2,4}$,$c_{3,1}$,$c_{4,1}$})	**344**	NS	355	4958
VCA(N, 3,5^7,CA(4,5^5))	Ct()	**827**	924	918	20250
	Ct({$c_{1,5}$,$c_{2,2}$,$c_{3,1}$,$c_{7,5}$}, {$c_{1,2}$,$c_{2,3}$,$c_{3,4}$}, {$c_{2,1}$,$c_{4,1}$,$c_{5,3}$,$c_{6,4}$}, {$c_{2,2}$,$c_{4,2}$,$c_{6,4}$,$c_{7,1}$}, {$c_{3,1}$,$c_{6,5}$,$c_{7,2}$}, {$c_{4,2}$,$c_{6,1}$,$c_{7,3}$})	**819**	NS	935	19994
VCA(N, 4,3^9,CA(5,3^6))	Ct()	**366**	381	377	8370
	Ct({$c_{1,3}$,$c_{4,1}$,$c_{7,2}$,$c_{9,1}$}, {$c_{2,2}$,$c_{3,3}$,$c_{6,1}$}, {$c_{4,1}$,$c_{5,1}$,$c_{6,3}$,$c_{8,2}$}, {$c_{5,2}$,$c_{6,2}$,$c_{8,2}$})	**357**	NS	378	8101
VCA(N, 4,4^9,CA(5,4^6))	Ct()	1505	1545	**1408**	90177
	Ct({$c_{1,1}$,$c_{5,3}$,$c_{6,4}$,$c_{8,2}$}, {$c_{2,2}$,$c_{3,4}$,$c_{4,1}$,$c_{7,2}$,$c_{9,1}$}, {$c_{1,2}$,$c_{2,1}$,$c_{8,4}$,$c_{9,4}$}, {$c_{3,2}$,$c_{4,3}$,$c_{6,1}$,$c_{7,4}$}, {$c_{1,3}$,$c_{2,4}$,$c_{3,1}$,$c_{4,3}$,$c_{5,2}$})	1500	NS	**1471**	87886
VCA(N, 4,5^9,CA(5,5^6)	Ct()	**4411**	5244	5187	536875
	Ct({$c_{2,2}$,$c_{3,4}$,$c_{8,5}$,$c_{9,5}$}, {$c_{3,3}$,$c_{4,3}$,$c_{7,4}$,$c_{8,2}$}, {$c_{2,1}$,$c_{5,3}$,$c_{6,5}$,$c_{7,4}$,$c_{8,1}$}, {$c_{6,2}$,$c_{7,1}$,$c_{8,4}$,$c_{9,1}$})	**4413**	NS	5186	534168
VCA(N, 4,$2^3 3^3 4^3$, CA(5,$2^2 3^2 4^2$))	Ct()	**337**	443	403	7512
	Ct({$c_{1,2}$,$c_{2,1}$,$c_{3,2}$,$c_{9,4}$}, {$c_{4,2}$,$c_{5,3}$,$c_{7,4}$,$c_{8,1}$}, {$c_{2,2}$,$c_{7,2}$,$c_{8,3}$,$c_{9,2}$}, {$c_{1,1}$,$c_{2,1}$,$c_{3,2}$,$c_{5,3}$,$c_{7,3}$}, {$c_{6,2}$,$c_{7,1}$,$c_{8,2}$,$c_{9,3}$})	**345**	NS	387	6938

6.0 CONCLUSION AND FUTURE WORKS

Summing up, this paper has proposed and evaluated a new variable strength t-way test suite generation strategy with constraints support. Presently, the support for variable strength t-way test suite generation with constraint support is still lacking in the literature. Owing to the need of many testing applications especially involving

software product lines (SPL)[39-41], the development of GVS_CONST represents the small leap in this direction.

Here, several conclusions can be made based on the evaluation results. First of all, empirical evidence demonstrates that GVS_CONSTis able to generate optimal test suite size for most system configurations (for both with and without constraints). Although some system configurations, GVS_CONST do not produce the optimal test suite size, the generated test suite size is competitive as compared to competing strategies.

Secondly, test suite with constraints does not necessary give smaller test suite than that of without constraints. Although theoretically test suite with constraints should be smaller that test suite without constraints (since number of tuples need to be covered is less due to constraints), however, the generated test suite size depends heavily on the strategy approach.

As for the future works, we are currently investigating possible enhancement for the GVS_CONST strategy to support higher interaction strength (i.e. $t > 4$). In addition, we are also considering integrating heuristic search in GVS_CONST strategy in order to further optimize the generated test suite size.

Acknowledgements

This research is funded by the generous MOHE ERGS grant "A Computational Strategy for Sequenced based T-Way Testing".

REFERENCES

[1] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "The Development of a Particle Swarm Based Optimization Strategy for Pairwise Testing," *Journal of Artificial Intelligence,* vol. 4, pp. 156-165, 2011.

[2] R. R. Othman and K. Z. Zamli, "T-Way Strategies and Its Applications for Combinatorial Testing," *International Journal of New Computer Architectures and their Applications* vol. 1, pp. 459-473, 2011.

[3] H. Y. Ong and K. Z. Zamli, "Development of Interaction Test Suite Generation Strategy With Input-Output Mapping Supports," *Scientific Research and Essays,* vol. 6, pp. 3418-3430, 2011.

[4] K. Z. Zamli, M. I. Younis, S. A. C. Abdullah, and Z. H. C. Soh, *Software Testing*. Kuala Lumpur: Open University Malaysia, 2008.

[5] M. F. J. Klaib, "Development Of An Automated Test Data Generation And Execution Strategy Using Combinatorial Approach," PhD, School of Electrical And Electronics, Universiti Sains Malaysia, 2009.

[6] K. Z. Zamli, M. F. J. Klaib, M. I. Younis, N. A. M. Isa, and R. Abdullah, "Design And Implementation Of A T-Way Test Data Generation Strategy With Automated Execution Tool Support," *Information Sciences,* vol. 181, pp. 1741-1758 2011.

[7] M. I. Younis, "MIPOG: A Parallel T-Way Minimization Strategy For Combinatorial Testing," PhD, School of Electrical And Electronics, Universiti Sains Malaysia, 2010.

[8] M. I. Younis and K. Z. Zamli, "MC-MIPOG: A Parallel T-Way Test Generation Strategy for Multicore Systems," *ETRI Journal,* vol. 32, pp. 73-83, 2010.

[9] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "MIPOG - Modification Of The IPOG Strategy For T-Way Software Testing," in *Proceeding of The Distributed Frameworks and Applications (DFmA)*, Penang, Malaysia, 2008.

[10] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "A Strategy For Grid Based T-Way Test Data Generation," in *Proceedings the 1st IEEE International Conference on Distributed Frameworks and Application (DFmA '08)*, Penang, Malaysia, 2008, pp. 73-78.

[11] A. W. Williams. (2010, February). *TConfig*. Available: http://www.site.uottawa.ca/~awilliam/

[12] A. W. Williams, "Determination of Test Configurations for Pair-wise Interaction Coverage," in *Proceedings of the 13th International Conference on Testing of Communicating System*, Ottawa, Canada, 2000, pp. 59-74.

[13] Y. W. Tung and W. S. Aldiwan, "Automatic Test Case Generation For The New Generation Mission Software System," in *Proceedings of IEEE Aerospace Conference*, Big Sky, MT, USA, 2000, pp. 431-437.

[14] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction Testing of Highly-Configurable Systems in the Presence of Constraints," in *International Symposium on Software Testing and Analysis (ISSTA2007)*, New York, USA, 2007, pp. 129-139.

[15] G. Sherwood. (2006, January). *Testcover*. Available: http://testcover.com

[16] R. R. Othman and K. Z. Zamli, "ITTDG: Integrated T-way Test Data Generation Strategy for Interaction Testing," *Scientific Research and Essays*, vol. 6, pp. 3638-3648, 2011.

[17] B. S. Ahmed, "Adopting a Particle Swarm-Based Test Generator Strategy for Variable-Strength and T-way Testing," PhD, School of Electrical And Electronics, Universiti Sains Malaysia, 2012.

[18] B. S. Ahmed and K. Z. Zamli, "A Variable Strength Interaction Test Suites Generation Strategy Using Particle Swarm Optimization," *Journal of Systems and Software*, vol. 84, pp. 2171-2185 2011.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy For T-Way Software Testing," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on The Engineering of Computer-Based Systems*, Tucson, AZ, 2007, pp. 549-556.

[20] Y. Lei, R. Kacker, R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation For Multi-Way Combinatorial Testing," *Journal of Software Testing, Verification and Reliability*, vol. 18, pp. 125-148, 2008.

[21] X. Chen, Q. Gu, A. Li, and D. Chen, "Variable Strength Interaction Testing With An Ant Colony System Approach," in *Proceedings of 16th Asia-Pacific Software Engineering Conference*, Penang, Malaysia, 2009, pp. 160-167.

[22] J. Czerwonka. (2010, February). *PICT Tool*. Available: http://www.pairwise.org/tools.asp

[23] J. Czerwonka, "Pairwise Testing In Real World," in *Proceedings of 24th Pacific Northwest Software Quality Conference*, Portland, Oregon, USA, 2006, pp. 419-430.

[24] Z. Wang, B. Xu, and C. Nie, "Greedy Heuristic Algorithms To Generate Variable Strength Combinatorial Test Suite," in *Proceedings of the 8th International Conference on Quality Software*, Oxford, UK, 2008, pp. 155-160.

[25] C. Nie and H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys*, vol. 43, 2011.

[26] M. B. Cohen, "Designing Test Suites For Software Interaction Testing," PhD, School of Computer Science, University of Auckland, 2004.

[27] L. Zekaoui, "Mixed Covering Arrays On Graphs And Tabu Search Algorithms," Master, Ottawa-Carleton Institute for Computer Science, University of Ottawa, Ottawa, Canada, 2006.

[28] A. Hartman and L. Raskin, "Problems and Algorithms for Covering Arrays," *Discrete Mathematics*, vol. 284, pp. 149-156, July 2004.

[29] A. S. Hedayat, N. J. A. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*. New York: Springer Verlag, 1999.

[30] R. Mandl, "Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing," *Communications of the ACM*, vol. 28, pp. 1054-1058, 1985.

[31] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, Tucson, AZ U.S.A, 2007, pp. 549-556.

[32] J. Yan and J. Zhang, "Backtracking Algorithms And Search Heuristics To Generate Test Suites For Combinatorial Testing," in *Proceeding of the 30th Annual International Computer Software and Applications Conference*, 2006, pp. 385-394.

[33] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach To Testing Based On Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, pp. 437-444, 1997.

[34] B. Jenkins. (2010, February). *Jenny Test Tool*. Available: http://www.burtleburtle.net/bob/math/jenny.html

[35] M. Grindal, J. Offutt, and S. F. Andler, "Combination Testing Strategies: A Survey," *Journal of Software Testing, Verification and Reliability*, vol. 15, pp. 167-199, 2005.

[36] R. C. Bryce and C. J. Colbourn, "A Density-Based Greedy Algorithm For Higher Strength Covering Arrays," *Software Testing, Verification and Reliability*, vol. 19, pp. 37-53, 2009.

[37] R. C. Bryce and C. J. Colbourn, "The Density Algorithm For Pairwise Interaction Testing," *Software Testing, Verification and Reliability*, vol. 17, pp. 159-182, 2007.

[38] Z. Wang, C. Nie, and B. Xu, "Generating Combinatorial Test Suite For Interaction Relationship," in *Proceedings of 4th International Workshop on Software Quality Assurance (SOQUA2007)*, Dubrovnik, Croatia, 2007, pp. 55-61.

[39] L. Yu, F. Duan, Y. Lei, R. Kacker, and K. Richard D., "Combinatorial Test Generation for Software Product Lines Using Minimum Invalid Tuples " in *Proceedings of the 15th IEEE Symposium on High-Assurance Systems Engineering*, 2014, pp. 66-72.

[40] J. Arshem. *Test Vector Generator Tool (TVG), available from http://sourceforge.net/projects/tvg, last accessed on March, 2010*.

[41] R. E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba, "Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 404 - 407.