MIPOG - An Efficient t-Way Minimization Strategy for Combinatorial Testing

Article · January 2011

DOI: 10.7763/IJCTE.2011.V3.337

CITATIONS
22

READS
70

2 authors:

Mohammed I. Younis
University of Baghdad
48 PUBLICATIONS **351** CITATIONS
SEE PROFILE

Kamal Z Zamli
Universiti Malaysia Pahang
167 PUBLICATIONS **1,250** CITATIONS
SEE PROFILE

MIPOG - An Efficient t-Way Minimization Strategy for Combinatorial Testing

Mohammed I. Younis and Kamal Z. Zamli

Abstract—**This paper presents a study comparing different techniques to achieve minimal test suites in combinatorial testing. Considering high interaction strength is not without difficulties. When the number of parameter coverage increases, the size of t-way test sets also increases exponentially, hence, resulting into combinatorial explosion problem. Addressing these aforementioned issues, a new strategy capable of supporting high interaction strength, called Modified IPOG (MIPOG) is proposed. Similar to its predecessor IPOG (In Parameter Order General), MIPOG adopts the horizontal and vertical extensions in order to construct the desired test set. However, unlike IPOG, MIPOG optimizes both the horizontal and vertical extensions resulting into a smaller size solution than that of IPOG, (i.e., with the test size ratio ≤ 1). In fact, MIPOG, in most cases, surpasses some IPOG variants (IPOD, IPOF1, and IPOF2) as well as other existing strategies (Jenny, TVG, TConfig, and ITCH), as far as the test size is concerned with an acceptable execution time. Additionally, MIPOG has also contributed to enhance many known CA and MCA that exist in the literature.**

Index Terms—**combinatorial testing, covering array, mixed covering array, multi-way testing, pairwise testing, *t*-way testing.**

I. Introduction

Testing is an important but expensive part of the software development process. Lack of testing often leads to disastrous consequences including loss of data, fortunes and even lives. For these reasons, many input parameters and system conditions need to be tested against the specifications of the system for conformance. Although desirable, exhaustive testing is prohibitively expensive even in a moderate-sized project, due to resources as well as timing constraints [1].Therefore, it is necessary to reduce the test selection space in a systematic manner. In line with increasing consumer demands for new functionalities and innovations, software applications grew tremendously in size over the last 15 years. This sudden increase has a profound impact as far as testing is concerned. Here, the test size grew significantly as a result. To address the aforementioned issues, much research is now focusing on sampling techniques based on interaction testing (termed *t-way testing strategy*) in order to derive the most optimum test suite for testing consideration (i.e., termed as Covering Array (CA) for uniform parameter values and Mixed Covering Array (MCA) for non-uniform parameter values respectively). Earlier adoption of t-way testing gave mixed results. While 2-way testing (also termed pairwise) testing appears to be adequate for achieving good test coverage in some existing system, a counter argument suggests that such a conclusion cannot be generalized to all (future) software system. Often, the net effect of software growth introduces new intertwined dependency between parameters involved, thus, justifying the need to support for high interaction strength (t).

One reduction approach is via pairwise testing [2, 3]. Pairwise testing helps detect faults caused by interactions between two parameters. Indeed, earlier work demonstrates that pairwise testing achieves higher block and decision coverage than traditional methods for a commercial email system [4]. While such a conclusion can be true for some system [5], a counter argument suggests that some faults may also be caused by the interaction of more than two parameters (i.e. often termed as t-way testing). For example, by applying t-way testing to a telephone software system demonstrates that several faults can only be detected under certain combinations of input parameters [6]. A study conducted by The National Institute of Standards and Technology (NIST) has shown that 95% of actual faults are caused by 4-way interactions in some system. In fact, only after considering up to 6-way interactions can all the faults be found [7, 8]. Given that software applications grew tremendously in the last 15 years, there is clearly a need to consider the support for high interaction strength, that is, to cater for the possibility of new intertwined dependencies between involved parameters.

Considering more than two parameter interactions is not without difficulties. When the number of parameter coverage increases, the size of t-way test sets also increases exponentially. As such, for a large system, considering a higher order t-way test set can lead to a combinatorial explosion problem. Here, computational efforts required in search of an optimum test set, termed as Covering Array (CA) (in which the parameter values are uniform) and Mixed Covering Array (MCA) (in which the parameter values are non-uniform), can be expensive especially when each interaction is to be covered optimally by the minimum number of test cases for a given interaction strength (t). Addressing the aforementioned issues, this paper proposes a new strategy, called Modified IPOG for t-way testing.

The remaining sections of this paper are organized as follows. Section 2 discusses some related work. Section 3

Manuscript received October 2, 2010; revised January 6, 2011. This work is partly sponsored by generous grants – "Development of Variable Strength Interaction Strategy for Combinatorial Test Data Generation", and Post Graduate Research Grant –"T-Way Test Data Generation Strategy Utilizing Multicore System" from Universiti Sains Malaysia.

Mohammed I. Younis with the Software Engineering Research Group of the School of Electrical and Electronic Engineering, USM (e-mail: younismi@gmail.com).

Kamal Z. Zamli with the Software Engineering Research Group of the School of Electrical and Electronic Engineering, USM (e-mail: eekamal@eng.usm.my).

gives the details of IPOG and our modified MIPOG. The similarities and differences between the two are also explained. Section 4 highlights comparisons between MIPOG and other existing tools. Finally, Section 5 gives the conclusions and suggestions for future work.

II. Related Work

Combinatorial testing has been used to generate test inputs for software testing. In combinatorial testing a set of inputs are modeled as factors and values and all combinations of each set of t-factors will be covered during testing using a structure called a covering a CA (when the values are equal) and MCA (for non-equal values). A number of strategies exist to cater for CA and MCA. In general, these strategies adopt either algebraic or computational approaches [9, 10].

Most algebraic approaches compute test sets directly by a mathematical function [11]. As the name suggests, algebraic approaches are often based on the extensions of the mathematical methods for constructing Orthogonal Arrays (OAs) [12, 13]. Younis et al. (2008) proposed a prime-based strategy to construct OA directly using a simple formula, and compared their strategy with the Latin OA strategy. Here, the prime strategy does not require any storage. Nonetheless, the prime strategy is restricted to the conditions that the number of parameters are equal to the number of values, and both are prime numbers [14].

Alternatively, OAs can be constructed by using Galois Finite field (GF) [15]. The limitation of this approach is two fold: the number of values (v) for each parameter should be a prime number or a power of prime number; and the number of parameters (p) should be $\leq$v.

Due to its popularity, catalogs of OAs can be found in the appendix of many advanced statistics books. In fact, Neil Sloan dedicates a web site to maintaining OA table [16]. While proven to be useful, OA-based approaches are often too restrictive (i.e. typically requiring the parameter values to be uniform). Furthermore, apart from merely supporting pairwise (or 2-way) interactions, not all OA solutions can be found for t>2.

An improvement of OA-based approaches, called CA has been developed as a result [17]. CA-based approaches are more flexible in the sense that it is independent of OA, i.e., a CA solution is possible to be found even without the existence of its OA solution. Some variations of the algebraic approach also exploit recursion in order to permit the construction of larger test sets from smaller ones [18]. In addition, test sets are derived without performing any explicit enumeration of the combinations to be covered. For instance, Maity et al. (2003) proposed a pairwise strategy for constructing CA by using algebraic product [19]. The computations involved in algebraic approaches are typically small and not subjected to the combinatorial explosion problem. For this reason, strategies that are based on algebraic approaches are extremely fast [20]. In some CA, algebraic strategies give the most optimum test suite size (i.e. within the lower bound) [21]. However, for large number of parameters, the upper bound can not be determined [22].

As an improvement of CA, MCA is proposed to cater for the support for non-uniform parameter values [23]. Maity and Nayak (2005) extend their CA strategy to generate some MCAs [24]. *Colbourn* et al. (2006) describe the construction of CAs and MCAs of Roux type [21]. In a nut shell, the construction of CAs or MCAs by means of pure algebraic approaches (i.e., without searching) can be achieved either by applying successive transformations to well known array or by using a product of construction [25, 26]. For this reason, algebraic approaches often impose restrictions on the system configurations to which they can be applied [1]. This significantly limits the applicability of algebraic approaches for software testing [20].

Unlike algebraic approaches, computational approaches often rely on the generation of all tuples and search the tuple space to generate the required test suite until all tuples have been covered [27]. In the case where the number of tuples to be considered is significantly large, adopting computational approaches can be expensive especially in terms of the space required to store the tuples and the time required for explicit enumeration. Unlike algebraic approaches, the computational approaches can be applied to arbitrary system configurations. Furthermore, computational approaches are more adaptable for constraint handling [28, 29] and test prioritization [30].

Hartman et al. (2005) developed the IBM's Intelligent Test Case Handler (ITCH) as an Eclipse Java plug-in tool [31]. ITCH uses a combinatorial algorithm based on exhaustive search to construct the test suites for *t*-way testing. Although useful as part of IBM's automated test plan generation, ITCH results appear to be not optimized as far as the number of generated test cases is concerned [20]. Furthermore, due to its exhaustive search algorithm, ITCH execution typically takes a long time. Concerning implementation, ITCH consists of two deterministic strategies [31] namely: CTS (Combinatorial Test Services) [10] and TOFU (Test Optimizer for Functional Usage) [32]. Both CTS and TOFU can support *t*-way test generation for $2\leq t\leq 4$. Typically, CTS performs better than TOFU in terms of test size and execution time.

Jenkins (2003) developed a deterministic *t*-way generation strategy, called Jenny [33]. Jenny adopts a greedy algorithm to produce a test suite in one-test-at-a time fashion. In Jenny, each feature has its own list of tuples. It starts out with 1-tuple (just the feature itself). When there are no tuples left to cover, Jenny goes to 2-tuples and so on. Hence, during generation instances, it is possible to have one feature still covering 2-tuples while another feature is already working on 3-tuples. This process goes on until all tuples are covered. Jenny has been implemented as an MSDOS tool using C programming language.

Complementary to the aforementioned work, significant efforts also involve extending existing pairwise strategies (e.g. in the case of *Automatic Efficient Test Generator* (AETG) and *In Parameter Order* (IPO)) to support *t*-way testing. AETG builds a test set "one-test-at-a-time" until all tuples are covered [2, 3]. In contrast, IPO covers "one-parameter-at-a-time" (i.e. through horizontal and vertical extension mechanisms), achieving a lower order of complexity than that of AETG [34].

Arshem (2003) developed a freeware Java-based *t*-way testing tool called Test Vector Generator (TVG) based on

extension of AETG to support *t*-way testing. Similar efforts are also undertaken by Bryce et al. (2005) to enhance AETG for *t*-way testing [35, 36]. Nie et al. (2005) proposed a generalization for IPO with the Genetic Algorithm (GA), called IPO_N, and GA_N, respectively for *t*=3. IPO_N performed better than GA_N in terms of test size as well as execution time [37].

Williams et al. (2003) implemented a deterministic Java-based *t*-way test tool called TConfig (Test Configuration) [38]. TConfig consists of two strategies, namely RE (REcursive algorithm); for *t*=2 [39, 40], and IPO for $2 \leq t \leq 6$ [38]. Williams reported that the RE failed to cover all tuples for *t*>2. For this reason, TConfig uses a minor version of IPO to cover the uncovered tuples in a greedy manner [15].

More recently, IPO is generalized to general *t*-way combinatorial testing into IPOG (In Parameter Order General)[41]. A number of variants have also been developed to improve the IPOG's performance. These variants including: IPOD [20], IPOF and IPOF2 [42].

Both IPOG and IPOD are deterministic strategies. Unlike IPOG, IPOD combines the IPOG strategy with an algebraic recursive construction, called D-construction developed by Chateauneuf et al (1999) [43], in order to reduce the number of tuples to be covered. In fact, Lei et al (2008) reported that when *t*=3, IPOD is degraded to the D-construction algebraic approach [20]. Here, when *t*>3, a minor version of IPOG is used to cover the uncovered tuples during D-construction [20]. As such, IPOD tends to be faster than IPOG, even with a high test size. It should be noted that the RE and IPO version used in TConfig differs from that used by IPOD.

Unlike IPOG and IPOD, IPOF is a non-deterministic strategy. For this reason, IPOF produces a different test set in each run. Unlike IPOG, IPOF rearranges the rows during the horizontal extension in order to cover more tuples per horizontal extension. Results on the performance of IPOF with a small number of parameter values have been reported in [42]. Similarly, a variant of IPOF, called IPOF2 [42] is also available, but it has been demonstrated with a small number of parameter values. Unlike IPOF, IPOF2 uses a heuristic technique to cover the tuples, allowing a faster execution time than that of IPOF but with a higher test set size. Currently, IPOG, IPOD, IPOF1 and IPOF2 are integrated into a Java-based tool called ACTS (Advanced Combinatorial Testing Suite). Finally, Cohen et al. (2008) proposed a heuristic search, particularly through the application of Simulated Annealing (SA) [44]. This local search method has provided many of the smallest test suites for different system configurations for t=2, and 3; however, at a cost in very high execution time to generate test suites [9, 44].

Overall, comparative results have shown that IPOG performed better than all the abovementioned *t*-way strategies (including some of its known variants such as IPOD, IPOF1, and IPOF2) particularly in terms of the support of higher order *t* with optimum test sizes and reasonable execution times [41]. For this reason, we have adopted the IPOG strategy as our benchmark.

As part of the effort to develop an optimized strategy for *t*-way testing, we have improved IPOG into a new strategy called Modified IPOG (MIPOG). Here, we aim to generate a more optimum test set, i.e. each *t*-way interaction is covered by as few test cases as possible; hence gives fewer combinations than that of IPOG. Furthermore, with MIPOG, we aim to contribute to the best well known results in [22]. It should be noted here that Colbourn collects the current best known upper bounds of CA for $2 \leq t \leq 6$ regardless of the strategies used (i.e. computational or algebraic approaches).

III. The Proposed Strategy

As discussed earlier, the proposed strategy, MIPOG, is based on the IPOG strategy [41]. For a system with at least *t* or more parameters, the MIPOG strategy constructs a *t*-way test set configuration for the first *t* parameters. Then, it extends the test set to construct a *t*-way test set for *t*+1 parameters. After that, it continues to extend the test set until a *t*-way test set has been constructed for all the parameters of the system. Like IPOG, MIPOG also performs the horizontal growth followed by the vertical growth, but in a different way in order to optimize the number of generated test sizes such that the *t*-way interaction element is covered by the minimum number of test cases. For comparative purposes, Fig. 1 and Fig. 2 illustrate the IPOG strategy and MIPOG strategy, respectively.

As can be seen in Fig.s 1 and 2, the inputs to both algorithms are the degree of interaction '*t*' and the set of parameters 'ps'. The output is a *t*-way test set for all the parameters in the system. The differences between the two strategies lie in both horizontal and vertical extensions (from line 6 onwards).

Algorithm *IPOG-Test (int t, ParameterSet ps)*
{
1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P1, P2, …, and Pn
3. add into test set ts a test for each combination of values of the first t parameters
4. for (int i = t + 1; i ≤ n; i ++){
5. let π be the set of t-way combinations of values involving parameter Pi and t -1 parameters among the first i – 1 parameters
6. // horizontal extension for parameter Pi
7. for (each test τ = (v1, v2, …, vi-1) in test set ts) {
8. choose a value vi of Pi and replace τ with τ' = (v1, v2, …, vi-1, vi) so that τ' covers the most number of combinations of values in π
9. remove from π the combinations of values covered by τ'
10. }
11. // vertical extension for parameter Pi
12. for (each combination σ in set π){
13. if (there exists a test that already covers σ) {
14. remove σ from π
15. } else {
16. change an existing test, if possible, or otherwise add a new test to cover σ and remove it from π
17. }
18. }
19.}
20. return ts;}

Fig. 1. The IPOG Strategy

Algorithm *MIPOG-Test (int t, ParameterSet ps)*
{
1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P1, P2, …, and Pn
3. add into test set ts a test for each combination of values of the first t parameters
4. for (int i = t + 1; i ≤ n; i ++){
5. let π be the set of t-way combinations of values involving parameter Pi and t -1 parameters among the first i – 1 parameters
6. // horizontal extension for parameter Pi
7. for (each test τ = (v1, v2, …, vi-1) in test set ts) {
8. if (τ not contains don't care){
// don't care means that there is a previous parameter(s) that //not assigned value(s). As
// such, it can be further optimized
choose a value vi of Pi and replace τ with τ' = (v1, v2,…, vi-1, vi) so that τ' covers the maximum number of combinations of values in π}
9. else {
choose a value vi of Pi and search all possible tuples that can be optimized the don't care to construct τ' = (v1, v2, …, vi-1, vi) so that τ' covers the maximum number of combinations of values in π and optimized the don't care
}
10. remove from π the combinations of values covered by τ'}
11. // vertical extension for parameter Pi
12. while (π not empty){
13. rearrange π in decreasing order according to the size of the remaining tuples
14. Choose the first tuple and generate test case (τ) that combine maximum number of tuples
15. delete the tuples covered by τ, add τ to local ts
16. } //while
17. return ts;}

Fig. 2. The MIPOG Strategy

In the horizontal extension, the MIPOG strategy checks all the values of the input parameters, and chooses the value that contains the maximum number of combinations for the uncovered tuples in the π set. MIPOG also optimizes the 'don't care' value. For this reason, MIPOG always generates a stable test case (that cannot be modified) by searching for tuples that can be covered by the same test. This is performed by means of searching of uncovered tuples that can be combined with the test case to fill the 'don't care' values during the horizontal extension (i.e. to ensure that the test case is indeed optimized).

In the vertical extension, MIPOG rearranges the π set in a decremented order size. After that, MIPOG chooses the first tuple from the rearranged π set and combines the tuple with other suitable tuples in the π set (i.e. the resulting test case must have the maximum weight of uncovered tuples). Once combined, all the tuples are removed from the π set. This process is repeated until the π set is empty (i.e. to ensure the complete interaction coverage).

To illustrate the differences between horizontal and vertical extensions of IPOG and MIPOG, we consider a system with 4 parameters (3 2-valued and 1 3-valued parameters). Fig. 3 and Fig. 4 demonstrate the processes of generating the 3-way test set for IPOG and MIPOG, respectively. Here, MIPOG generates a minimal test set (3*2*2=12 values), while IPOG generates 14 test cases.

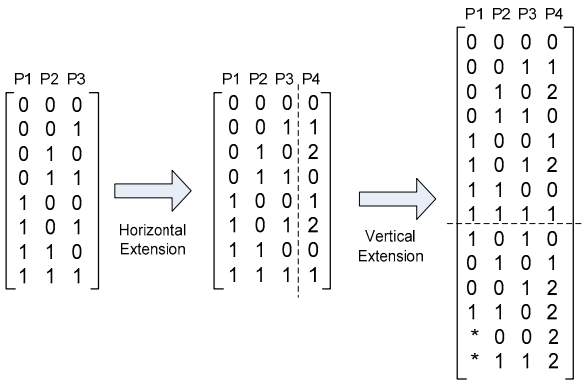

Fig. 3. Generation of Test Set Using IPOG

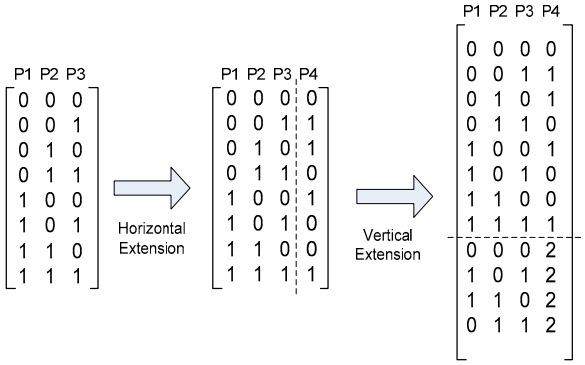

Fig. 4. Generation of Test Set Using MIPOG

Because of the optimization process in the MIPOG strategy (i.e. in search for the optimized tuples to be combined with the current test case in the vertical and horizontal extensions), we expect the MIPOG strategy to always give the same or a smaller test set than that of IPOG (especially in the case involving the 'don't care' values). However, such an optimization process is not without a cost. We do expect that the MIPOG strategy to be slightly slower than that of IPOG as far as the execution time is concerned.

IV. Evaluation and Discussion

In this section, we evaluate MIPOG with the following objectives:

i. To investigate the overall performance of MIPOG
ii. To investigate whether MIPOG can have a significant gain against IPOG in terms of test size ratio.
iii. To investigate whether MIPOG contributes to the best known results for CAs.
iv. To compare against other existing strategies especially for MCAs.

In order to achieve the first three objectives, three groups of experiments are applied to determine the respective CAs adopted from Lei et al (2008) [20].

- In the first group, we fix both the number of parameters (p) to 10 and the strength of coverage (t) to 5, and vary the values (v) from 2 to 7.

- In the second group, we fix both (v) and (t) to 5, and change (p) from 6 to16.
- In the third group, we fix p=10 and v=6, and change t from 2 to 7.

To perform the experiments, we have downloaded ACTS from the NIST website [45]. In this case, the comparison is fair since MIPOG and IPOG (which is an available option in ACTS tool) are executed within the same platform consisting of Windows XP, with 1.6 GHz CPU, 1 GB RAM, and with JDK 1.6 installed. The results of the three groups of experiments are tabulated in Tables 1, 2, and 3, respectively. Darkened cells indicate the best performance in term of the test size and the execution time. Entries marked 'NS' indicate non-supported features of the corresponding tool for the corresponding value of t.

TABLE 1: RESULTS FOR CAS WITH T= 5, P=10, AND V=(2,..,7)

V	MIPOG		IPOG		MIPOG/IPOG Size Ratio
	Size	Time (second)	Size	Time (second)	
2	92	0.172	98	0.221	0.939
3	626	0.5 93	751	1.361	0.834
4	2911	5.046	3057	6.375	0.952
5	8169	45.95	10111	31.875	0.879
6	23557	766.58	25247	126.156	0.933
7	49597	1365.45	54567	430.937	0.909

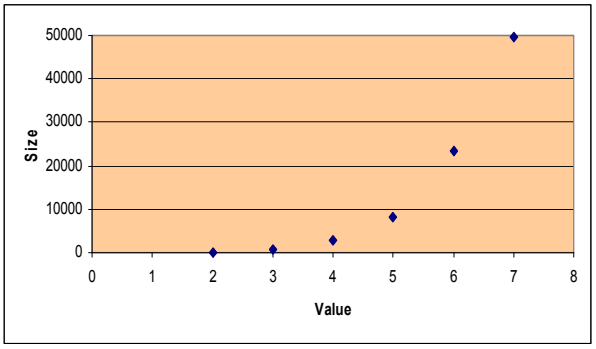

Fig. 5. MIPOG's Test Size versus Values for Group 1

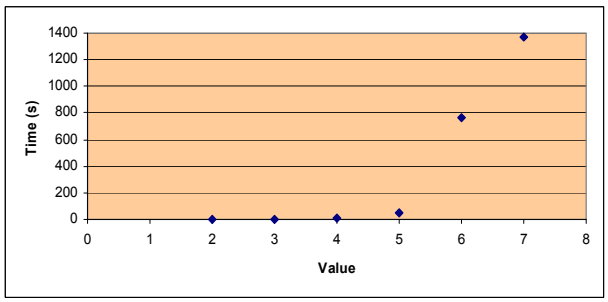

Fig. 6. MIPOG's Execution Time versus Value for Group 1

From Table 1, we plot the test size against the number of values, as given in Fig. 5. We also plot the execution time versus the number of values, as shown in Fig. 6.

Referring to Figures 5 and 6, we conclude that both the test size and execution time are proportional quinary with the number of values.

From Table 2, we plot the test size versus the number of parameters, as given in Fig. 7. Then, we also plot the execution time versus the number of parameters, as shown in Fig. 8.

TABLE 2: RESULTS FOR CAS WITH T= 5, P=(6,..,16), AND V=5

P	MIPOG		IPOG		MIPOG/IPOG Size Ratio
	Size	Time (second)	Size	Time (second)	
6	3125	0.97	4149	1.265	0.753
7	5625	2.86	6073	4.656	0.926
8	5954	12.569	7517	8.796	0.792
9	6996	24.462	8882	27.656	0.787
10	8169	57.444	10111	31.875	0.808
11	9067	147.488	11276	55.594	0.804
12	9974	330.740	12337	107.266	0.808
13	11004	1112.302	13361	195.86	0.824
14	11924	4476.252	14284	334.829	0.835
15	12704	13881.09	15168	528.907	0.837
16	13469	31577.4	15993	848.062	0.842

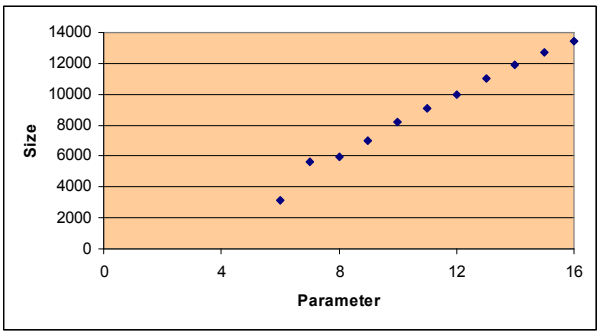

Fig. 7. MIPOG's Test Size versus Parameter for Group 2

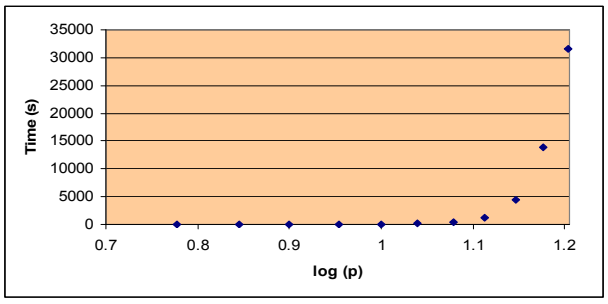

Fig. 8. MIPOG's Execution Time versus log (p) for Group 2

Referring to Fig. 7, we conclude that the test size grows logarithmically with the increasing. number of parameters. From Fig. 8, we note that the execution time grows in quinary with respect to the logarithmic scale of parameters.

In order to investigate the characteristics of the test size and execution time against varying strength of coverage (t), we plot the test size versus t from Table 3, as given in Fig. 9. Here, we also plot the execution time versus t, as shown in Fig. 10.

TABLE 3: RESULTS FOR CAS WITH T=(2,..,7), P=10, AND V=6

t	MIPOG		IPOG		MIPOG/IPOG Size Ratio
	Size	Time (second)	Size	Time (second)	
2	63	0.33	67	0.073	0.94
3	512	1.797	532	0.594	0.962
4	3657	69.05	3843	8.945	0.951
5	23557	766.58	25247	126.156	0.933
6	139638	13239.92	152014	2773.485	0.918
7	775163	46259.89	NS		-

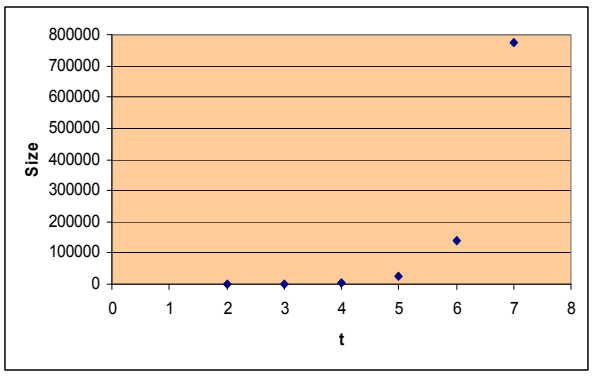

Fig. 9. Test Size versus (t) for Group 3

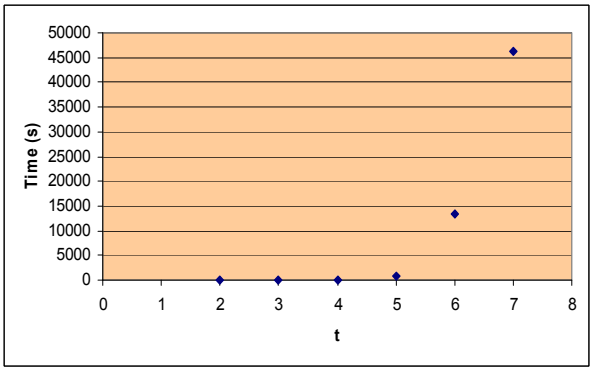

Fig. 10. Execution Time versus (t) for Group 3

From Figures 9 and 10, it is evident that the test sizes as well as the execution time grow exponentially as the strength of coverage (t) increases. Putting all together, the test size as well as the execution time can be summoned in $O(v^t \log p)$.

In order to achieve the second objective, we re-visit the size ratio in the last column of Tables 1, 2, and 3, respectively. Here, the size ratio is defined as the size obtained from MIPOG to the size obtained from IPOG. From these tables, we note that the size ratio is always ≤1, which is an indication that MIPOG is able to outperform IPOG in terms of the test size. On one hand, as expected, due to optimization of both horizontal and vertical extensions, we observe that IPOG in most cases outperformed MIPOG in terms of the execution time. On the other hand, MIPOG's test size is significantly smaller than that of IPOG. Furthermore, MIPOG supports t≥6 whilst IPOG support up to t=6 (see Table 3).

To achieve the third objective, we compare our results against the reported results in Colbourn's catalogue [22]. Here, we adopt the Colbourn's notation CA (t,p,v) where t indicates the interaction strength, p indicates the number of parameters, and v indicates the number of values. Table 4 reports the new upper bound of the existing CAs (i.e. CA(4,6,6), CA(5,8,3), CA(5,8,5), CA(5,9,5), CA(5,10,5) CA(5,11,5), CA(5,13,5), CA(5,14,5), CA(5,15,5), CA(5,16,5), CA(5,7,6), CA(6,8,6), CA(6,9,6), and CA(6,10,6)) as well as new ones (CA(7,8,6), CA(7,9,6), and CA(7,10,6)).

In order to achieve the final objective, we consider non-homogeneous (i.e., mixing) MCAs. We have downloaded all the tools (IPOG, IPOD, IPOF, IPOF2, TVGII, Jenny, and TConfig) within our platform. Note that IPOG, IPOD, IPOF, and IPOF2 are integrated in the ACTS tool. Here, we subject MIPOG and all other tools to a series of experiments using our in-house RFID Tracking System module (TS). As explained earlier, the TS module has eleven parameters: seven parameters have 5 values, and four parameters have 2 values. The results are tabulated in Tables 5 and 6. Similar to the earlier results (see Tables 1 to 3), darkened cells indicate the best performances in terms of the test size and the execution time. Entries marked 'NA' indicate that the results are not available (even though the tool supports the data entry, the execution time exceeded one day (i.e. > 1 day). Entries marked 'NS' indicate non-supported features of the corresponding tool.

TABLE 4: NEW CANS DERIVED FROM MIPOG

CAN (t,p,v)	MIPOG	Best Upper Bound [22]	
	Size(N)	Algorithm	Old Size
CA(4,6,6)	1851	Soriano CA(4,7,6)	1893
CA(5,8,3)	432	Simulated Annealing (Cohen)	457
CA(5,8,5)	5954	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	6392
CA(5,9,5)	6996	Density (Colbourn algorithm by Linnemann-Frewer) postop Nayeri-Colbourn-Konjevod	7647
CA(5,10,5)	8169	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	8555
CA(5,11 ,5)	9067	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	9793
CA(5,13 ,5)	11004	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	11944
CA(5,14 ,5)	11924	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	12777
CA(5,15 ,5)	12704	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	14326
CA(5,16 ,5)	13469	Density (Colbourn) postop Nayeri-Colbourn-Konjevod	14326
CA(5,7,6)	12944	IPOF (NIST)	14712
CA(6,8,6)	87818	CA(6,7,6) extends by one factor	103446
CA(6,9,6)	115811	CA(6,8,6) extends by one factor	160236
CA(6,10,6)	139638	Composition	208656
CA(7,8 ,6)	279936	NA	NA
CA(7,9 ,6)	569050	NA	NA
CA(7,10 ,6)	775163	NA	NA

TABLE 5: MCAS FOR TS MODULE WHERE T CHANGES FROM 2 TO 11

t	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig		ITCH	
										CTS	TOFU
								RE	IPO		
2	**38**	41	52	39	40	41	43	45	40	45	121
3	**218**	239	277	240	244	245	270	239		225	1358
4	**1154**	1290	1850	1262	1311	1273	1420	1320		1750	NA
5	**5625**	6073	9894	5975	6036	6268	6501	NA		NS	NS
6	**17527**	21452	33611	22135	22485	26012	25601	NA		NS	NS
7	**78940**	NS	NS	NS	NS	NA	NA	NS		NS	NS
8	**158526**	NS	NS	NS	NS	NA	NA	NS		NS	NS
9	**468750**	NS	NS	NS	NS	NA	NA	NS		NS	NS
10	**625000**	NS	NS	NS	NS	NA	NA	NS		NS	NS
11	**1250000**	NS	NS	NS	NS	NA	NA	NS		NS	NS

TABLE 6: TIME REQUIRED TO GENERATE MCAS FOR TS MODULE WHERE T CHANGES FROM 2 TO 11

t	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig		ITCH	
										CTS	TOFU
2								RE	IPO		
	0.07	0.016	**<0.001**	0.01	0.01	0.11	0.05	0.11	0.1	0.55	2.3
3	0.275	0.188	**0.002**	0.02	0.031	0.5	2.281	25.32		1.23	30.44
4	1.125	1.75	**0.188**	0.422	0.563	3.09	29.66	>10hour		120.665	>day
5	35.15	17.891	**4.375**	7.502	10.1	66.41	2164.98	>day		NS	NS
6	350.12	68.531	**59.016**	114.33	128.64	842.33	>9hour	>day		NS	NS
7	**680.23**	NS	NS	NS	NS	>day	>day	NS		NS	NS
8	**990.77**	NS	NS	NS	NS	>day	>day	NS		NS	NS
9	**1800.82**	NS	NS	NS	NS	>day	>day	NS		NS	NS
10	**3399.66**	NS	NS	NS	NS	>day	>day	NS		NS	NS
11	2.483	NS	NS	NS	NS	>day	>day	NS		NS	NS

Referring to Table 5, MIPOG clearly outperforms all existing strategies in terms of the test size. Furthermore, MIPOG appears to be the only strategy that generate a test suite for t>6. As seen in Table 7, for low value of t, IPOD has the fastest time. For t>7, MIPOG outperforms all other strategies. Even though Jenny and TVGII accept the request of generation of a test suite for t>6, no results were obtained after 1 day (see 'NS' and 'NA' entries in Tables 5 and 6, respectively). Another observation is that the execution time for MIPOG for 10-way test data generation is more than 11-way (i.e., an exhaustive testing). Such a result is expected as there is no need for optimization in the case of exhaustive testing; thus, rendering faster computation.

Going back to Table 6, the fact that IPOD (as well as other IPOG variants) is dominant as far as the execution time is concerned is justifiable. The general aim of IPOD is to get a faster execution time than that of its predecessor, IPOG. In general, getting an optimized test size and obtaining a fast execution time are two complementary and intertwined issues. On one hand, it is desirable to achieve a fast execution time under the cost of little optimization as far as the test size is concerned. On the other hand, obtaining the most minimum test size typically requires a longer execution time in order to select the most optimum interaction elements. As discussed earlier, MIPOG adopts different horizontal and vertical extension mechanisms; that requires more computation (i.e., in order to optimize the 'don't care') than that of IPOG (including IPOD and IPOF). Thus, MIPOG's execution time tends to be slower than those from most of the IPOG family, a reasonable cost to pay for smaller test sizes.

As far as investigating the effects of variations in domain sizes is concerned, we have also applied all the strategies to four system configurations with mixed domain sizes (i.e. similar to previous studies. Tables 7 to 11 depict the results for the four configurations for t=2, 3, 4, 5, and 6, respectively, in terms of the test size. Column "Configuration" shows the parameters and values of each configuration in the following format: $d_1^{k1}\ d_2^{k2}$ indicate that there are $k1$ parameters with $d1$ values, $k2$ parameters with $d2$ values, and so on. For example, configuration $5^1 3^8 2^2$ in the second row indicates that there is one parameter with five values, eight parameters with three values, and two parameters with two values. In addition to the earlier defined entry of 'NA', entries marked '-' in Table 7 indicate that no best result is published. The last column in Table 7 indicates the best published results by Bryce et al. [46]. Here, the column enteries "1C", "5C", and "10C" refers to using one candidate, five candidates, and 10 candidates test cases respectively. During the searching process, then select the test case (from these candidates) that covers the maximum number of uncovered tuples.

Referring to Tables 7 to 11, MIPOG outperforms all other strategies for most cases, except for the third row in the Configuration column. Here, OA exists for 9-8 valued parameters algebraically. As such, the third row is a subset of this OA. Both CTS in ITCH and RE in TConfig are able to produce the OA (see Table 7). The same observation is applicable to CTS in the case of t=3 (see Table 8).

IPOF gives the minimal test suite for t=4 (see Table 9) using the dynamic programming technique [42] rather than algebraic. Additionally, IPOF produces the minimal test suite for the first and third configurations for t=5 (see Table 10). MIPOG produces the minimal test suite when t=6 (see Table 11).

From the overall results, two conclusions can be drawn. Firstly, even though an algebraic OA exists, the algebraic strategy will not necessarily produce the minimal test suite in the case of MCAs for varying t. For the case of CA, the algebraic strategy always gives the minimal test suite when OA exists.

Finally, keeping MIPOG aside, IPOF, IPOF2, IPOG, IPOD, and Jenny are suitable for generating test suites for $t \leq 6$ within acceptable times. TVG II also could produce a test suite for t=6, for some configurations. Similarly, TConfig could produce a test suite for t=5, for some configurations. TOFU is more suitable for generating test suites for t=2, and t=3, and only supports t=4 for small system configurations (see Table 9). In most cases, TOFU and CTS require a longer execution time to produce the required test suite.

TABLE 7: COMPARATIVE RESULTS FOR FOUR SYSTEM CONFIGURATIONS WITH MIXED DOMAIN SIZES USING T=2

Configuration	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig		ITCH		Best Published [46]		
								RE	IPO	CTS	TOFU	1C	5C	10C
$4^5 3^4$	**22**	24	31	24	24	26	27	28	**22**	28	75	23	23	**22**
$5^1 3^8 2^2$	**17**	19	29	23	22	23	22	21	19	41	31	20	19	19
$8^2 7^2 6^2 5^2$	67	73	112	69	70	76	79	**64**	78	**64**	231	69	68	68
$10^2 4^1 3^2 2^7$	**100**	**100**	130	**100**	**100**	106	101	120	**100**	120	132	-	-	-

TABLE 8: COMPARATIVE RESULTS FOR FOUR SYSTEM CONFIGURATIONS WITH MIXED DOMAIN SIZES USING T=3

Configuration	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig	ITCH	
									CTS	TOFU
$4^5 3^4$	**101**	108	121	103	114	115	122	103	112	593
$5^1 3^8 2^2$	**78**	85	113	85	87	85	88	89	222	577
$8^2 7^2 6^2 5^2$	558	591	729	560	623	645	716	594	**511**	3031
$10^2 4^1 3^2 2^7$	**400**	**400**	480	402	427	411	434	472	2415	471

TABLE 9: COMPARATIVE RESULTS FOR FOUR SYSTEM CONFIGURATIONS WITH MIXED DOMAIN SIZES USING T=4

Configuration	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig	ITCH	
									CTS	TOFU
$4^5 3^4$	**425**	434	704	429	455	452	481	444	704	NA
$5^1 3^8 2^2$	**275**	300	527	291	317	303	317	302	1683	3518
$8^2 7^2 6^2 5^2$	4163	4302	7571	**4077**	4491	4580	5098	4317	4085	NA
$10^2 4^1 3^2 2^7$	**1265**	1361	2522	1352	1644	1527	1599	1476	1484	NA

TABLE 10: COMPARATIVE RESULTS FOR FOUR SYSTEM CONFIGURATIONS WITH MIXED DOMAIN SIZES USING T=5

Configuration	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig
$4^5 3^4$	1562	1625	2859	**1561**	1642	1667	1699	1641
$5^1 3^8 2^2$	**901**	983	1909	960	1015	996	1005	986
$8^2 7^2 6^2 5^2$	26023	27676	42380	**25954**	27995	29326	31707	NA
$10^2 4^1 3^2 2^7$	**4196**	4219	5306	4290	5018	4680	4773	NA

TABLE 11: COMPARATIVE RESULTS FOR FOUR SYSTEM CONFIGURATIONS WITH MIXED DOMAIN SIZES USING T=6

Configuration	MIPOG	IPOG	IPOD	IPOF	IPOF2	Jenny	TVG II	TConfig
$4^5 3^4$	**4972**	5393	8143	5125	5331	5501	5495	NA
$5^1 3^8 2^2$	**2657**	2910	5179	2844	2971	3017	3100	NA
$8^2 7^2 6^2 5^2$	**141445**	154315	170098	144510	151973	179591	>day	NA
$10^2 4^1 3^2 2^7$	**10851**	10919	14480	11234	13310	11608	12732	NA

V. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an efficient t-way test data generator (MIPOG) based on IPOG. Like IPOG, MIPOG adopts the horizontal and vertical extensions in order to construct the desired test set. Unlike IPOG, MIPOG optimizes both the horizontal and vertical extensions resulting in a smaller size solution than that of IPOG (i.e. with the test size ratio of ≤ 1). In fact, MIPOG, in most cases, surpasses other existing strategies (Jenny, TVG, TConfig, ITCH) including some known IPOG and its variants (IPOD, IPOF1, IPOF2), as far as the test size is concerned with an acceptable execution time. As such, MIPOG has also contributed to enhance many known CA and MCA as described in the literature.

Considering the fact that in most cases MIPOG generates the small test suite sizes (for both CA and MCA) with an acceptable execution time, our evaluation of MIPOG is encouraging. In fact, our experience also indicates MIPOG is capable to generate higher strength test suite that has never

been reported in the literature (i.e., t>6). As part of future work, we are integrating MIPOG within the GRID environment in order to obtain more speed as far as execution time is concerned. In addition, we will improve the algorithm to handle other practical testing issues such as dependencies between factors and values.

ACKNOWLEDGMENT

Here, we would like to thank all the anonymous reviewers for giving a comprehensive review on our paper. Special thanks to Prof. C.P. Lim for his valuable efforts and comments to improve the paper. In addition, we acknowledge the help of Jeff Lei, Raghu Kacker, Rick Kuhn, Myra B. Cohen, and Bob Jenkins for providing us with useful comments and the background materials.

REFERENCES

[1] M. I. Younis, K. Z. Zamli, M. F. J. Klaib, Z. C. Soh, S. C. Abdullah, and N. A. M. Isa, "Assessing IRPS as an Efficient Pairwise Test Data Generation Strategy," International Journal of Advanced Intelligence Paradigms (IJAIP), vol. 2, pp. 90-104, 2010.

[2] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, pp. 83-88, 1996.

[3] 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. 437–443, 1997.

[4] K. Burr and W. Young, "Combinatorial Test Techniques: Table Based Automation, Test Generation and Code Coverage," in Proceedings of the International Conference on Software Testing Analysis & Review (STAR), San Diego, CA, 1998, pp. 503-513.

[5] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," IEEE Transactions on Software Engineering, vol. 31, pp. 20–34, 2006.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-Based Testing in Practice," in International Conference on Software Engineering, Los Angeles, California, United State, 1999, pp. 285 -294.

[7] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions on Software Engineering, vol. 30, pp. 418-421, 2004.

[8] D. R. Kuhn and V. Okun, "Pseudo Exhaustive Testing For Software," in Proceedings of the 30th NASA/IEEE Software Engineering Workshop, Washington, DC, USA, 2006, pp. 153-158.

[9] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing Test Suites for Interaction Testing," in Proceedings of the 25th IEEE International Conference on Software Engineering, Portland, Oregon, 2003, pp. 38-48.

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

[11] M. Grindal, J. Offutt, and S. Andler, "Combination Testing Strategies: a Survey," Software Testing, Verification, and Reliability, vol. 15, pp. 167-199, 2005.

[12] K. A. Bush, "Orthogonal Arrays of Index Unity," Annals of Mathematical Statistics, vol. 23, pp. 426-434, 1952.

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

[14] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "Algebraic Strategy to Generate Pairwise Test Set for Prime Number Parameters and Variables," in Proceedings of the 3rd International Symposium on Information Technology (ITSim'08), KLCC, Malaysia, 2008, pp. 1662-1666.

[15] A. W. Williams, "Software Component Interaction Testing: Coverage Measurment and Generation of the Configurations (PhD Thesis)," in School of Information Technology and Engineering Ottawa, Canada: University of Ottawa, 2002.

[16] N. J. A. Sloane, "A Library of Orthogonal Arrays," Information Sciences Research Center, AT&T Shannon Labs, available from http://www.research.att.com/~njas/oadir, last accessed on March, 2010.

[17] B. Stevens and E. Mendelsohn, "Efficient Software Testing Protocols," in Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON '98), Toronto, Ontario, Canada, 1998, pp. 279-293.

[18] A. W. Williams, "Determination of Test Configurations for Pair-Wise Interaction Coverage," in Proceedings of the 13th International Conference on the Testing of Communicating Systems (Testcom 2000), Ottawa, Canada, 2000, pp. 59-74.

[19] S. Maity, A. Nayak, M. Zaman, N. Bansal, and A. Srivastav, "An Improved Test Generation Algorithm for Pair-Wise Testing," in Proceedings of the 14th International Symposium on Software Reliability Engineering (Fast Abstract ISSRE 2003) Denver, Colorado: Chillarege Press, 2003.

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

[21] C. J. Colbourn, S. S. Martirosyan, T. Trung, and R. A. Walker, "Roux-Type Constructions for Covering Arrays of Strengths Three and Four," Designs, Codes, and Cryptography, vol. 41, pp. 33-57, 2006.

[22] C. J. Colbourn, "Covering Array Tables, available from http://www.public.asu.edu/ccolbou/src/tabby, last access on March 2010."

[23] M. B. Cohen, "Designing Test Suites for Software Interaction Testing (PhD Thesis)," in Computer Science Auckland: University of Auckland, 2004.

[24] S. Maity and A. Nayak, "Improved Test Generation Algorithms for Pairwise Testing," in Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), Chicago, Illinois, USA, 2005, pp. 235-244.

[25] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas, "Products of Mixed Covering Arrays of Strength Two," Journal of Combinatorial Designs, vol. 14, pp. 124–138, 2006.

[26] G. B. Sherwood, "Pairwise Testing Comes of Age," Testcover Inc., 2008.

[27] M. I. Younis, K. Z. Zamli, and N. A. M. Isa, "IRPS: An Efficient Test Data Generation Strategy for Pairwise Testing," in Proceedings of the 12th international conference on Knowledge-Based Intelligent Information and Engineering Systems, Part I, Zagreb, Croatia, 2008, pp. 493-500.

[28] M. Grindal, J. Offutt, and J. Mellin, "Conflict Management when Using Combination Strategies for Software Testing," in Proceedings of the 18th Australian Software Engineering Conference (ASWEC 2007), Melbourne, Australia, 2007, pp. 1-10.

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

[30] R. C. Bryce and C. J. Colbourn, "Prioritized Interaction Testing for Pairwise Coverage with Seeding and Avoids," Information and Software Technology Journal, vol. 48, pp. 960-970, 2006.

[31] A. Hartman, T. Klinger, and L. Raskin, "WHITCH: IBM Intelligent Test Configuration Handler," IBM Haifa and Watson Research LaboratoriesApril 2005 2005.

[32] R. Biyani and P. Santhanam, "TOFU Test Optimizer for Functional Usage," Software Engineering Technical Brief, IBM T.J. Watson Research Center, vol. 2, 1997.

[33] B. Jenkins, "Jenny Test Tool, available from http://www.burtleburtle.net./bob/math/jenny.html, last accessed on April, 2010.."

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

[35] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Test Suites," in Proceedings of the 27th IEEE International Conference on Software Engineering, NY, USA, 2005, pp. 146-155.

[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] C. Nie, B. Xu1, L. Shi1, and G. Dong, "Automatic Test Generation for N-Way Combinatorial Testing," LNCS, vol. 3712, pp. 203-211, Friday, September 09, 2005 2005.

[38] A. W. Williams, J. H. Ho, and A. Lareau, "TConfig Test Tool Version 2.1,"Ottawa, Ontario, Canada, available from http://www.site.uottawa.ca/~awilliam, last access on March 2010:

School of Information Technology and Engineering (SITE), University of Ottawa, 2003.
[39] A. W. Williams and R. L. Probert, "A Practical Strategy for Testing Pair-Wise Coverage of Network Interfaces," in Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, New York, 1996, pp. 246-254.
[40] A. W. Williams and R. L. Probert, "A Measure for Component Interaction Test Coverage," in Proceedings of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001), Beirut, Lebanon, 2001, pp. 304-311.
[41] 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 (ECBS2007), Tucson, AZ, 2007, pp. 549-556.
[42] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays," Journal of Research of the National Institute of Standards and Technology, vol. 113, pp. 287-297., October 2008 2008.
[43] M. A. Chateauneuf, C. J. Colbourn, and D. L. Kreher, "Covering Arrays of Strength 3," Designs, Codes, and Cryptography, vol. 16, pp. 235-242, 1999.
[44] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Constructing Strength Three Covering Arrays with Augmented Annealing," Discrete Mathematics, vol. 308, pp. 2709-2722, 2008.
[45] NIST, "Web Site, Automated Combinatorial Testing for Software, available from http://csrc.nist.gov/groups/SNS/acts, last accessed on September, 2010."
[46] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A Framework of Greedy Methods for Constructing Interaction Test Suites," in Proceedings of 27th IEEE International Conference on Software Engineering, NY, USA, 2005, pp. 146-155.

Mohammed I. Younis obtained his BSc in computer engineering from the University of Baghdad in 1997, and his MSc degree from the same university in 2001, and PhD in software engineering and parallel processing from Universiti Sains Malaysia in 2010. He is currently a Post-Doc researcher attached to the Software Engineering Research Group of the School of Electrical and Electronic Engineering, USM.

He is a Senior Lecturer and a Cisco instructor at Computer Engineering Department, College of Engineering, University of Baghdad. He is also a software-testing expert in Malaysian Software Engineering Interest Group (MySEIG). His research interests include software engineering, parallel and distributed computing, algorithm design, RFID, networking, and security. Dr. Younis is also a member of Iraqi Union of Engineers, IEEE, IET, *IAENG,* CEIA, and ICCIS.

Kamal Z. Zamli obtained his BSc in Electrical Engineering from Worcester Polytechnic Institute,Worcester, USA in 1992, MSc in Real Time Software Engineering from CASE, Universiti Teknologi Malaysiain 2000, and PhD in Software Engineering from the University of Newcastle upon Tyne, UK in 2003. He is currently an Associate Professor attached to the Software Engineering Research Group, in the School of Electrical and Electronic Engineering, USM.

He is also a software-testing expert in Malaysian Software Engineering Interest Group (MySEIG). His research interests include software engineering, software testing automation, and algorithm design. Dr. Zamli is also a member in IEEE, and IET.