G2Way A Backtracking Strategy for Pairwise Test Data Generation

Conference Paper · January 2008

DOI: 10.1109/APSEC.2008.49 · Source: DBLP

5 authors, including:

Mohammad F. J. Klaib
Jadara University
7 PUBLICATIONS **99** CITATIONS
SEE PROFILE

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

Nor Ashidi Mat Isa
Universiti Sains Malaysia
225 PUBLICATIONS **2,398** CITATIONS
SEE PROFILE

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

G2Way - A Backtracking Strategy for Pairwise Test Data Generation

Mohammad F. J. Klaib, Kamal Z. Zamli, Nor Ashidi M. Isa, and Mohammed I. Younis
School of Electrical and Electronics
Universiti Sains Malaysia
14300 Nibong Tebal, Penang, Malaysia
Email: {eekamal, ashidi}@eng.usm.my

Rusli Abdullah
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia,
43400 Serdang, Selangor, Malaysia
Email: rusli@fsktm.upm.edu.my

Abstract

Our continuous dependencies on software (i.e. to assist as well as facilitate our daily chores) often raise dependability issue particularly when software is being employed harsh and life threatening or (safety) critical applications. Here, rigorous software testing becomes immensely important. Many combinations of possible input parameters, hardware/software environments, and system conditions need to be tested and verified against for conformance. Due to resource constraints as well as time and costing factors, considering all exhaustive test possibilities would be impossible (i.e. due to combinatorial explosion problem). Earlier work suggests that pairwise sampling strategy (i.e. based on two-way parameter interaction) can be effective. Building and complementing earlier work, this paper discusses an efficient pairwise test data generation strategy, called G2Way. In doing so, this paper demonstrates the correctness of G2Way as well as compares its effectiveness against existing strategies including AETG and its variations, IPO, SA, GA, ACA, and All Pairs. Empirical evidences demonstrate that G2Way, in some cases, outperformed other strategies in terms of the number of generated test data within reasonable execution time.

1. Introduction

Nowadays, we are increasingly dependent on software to assist as well as facilitate our daily chores. In fact, whenever possible, most hardware implementation is now being replaced by the software counterpart. From the washing machine controllers, mobile phone applications to the sophisticated airplane control systems, the growing dependent on software can be attributed to a number of factors. Unlike hardware, software does not wear out. Thus, the use of software can also help to control maintenance costs. Additionally, software is also malleable and can be easily changed and customized as the need arises.

Our continuous dependencies on software often raise dependability issue particularly when software is being employed harsh and life threatening or (safety) critical applications. Here, rigorous software testing becomes immensely important. Many combinations of possible input parameters, hardware/software environments, and system conditions need to be tested and verified against for conformance based on the system's specification. Often, this results into combinatorial explosion problem.

Combinatorial explosion problem [3, 18] poses one of the biggest challenges in modern computer science due to the fact that it often defies traditional approaches to analysis, verification, monitoring and control. A number of techniques have been explored in the past to address this problem. Undoubtedly, parallel testing can be employed to reduce the time required for performing the tests. Nevertheless, as software and hardware are getting more complex than ever, parallel testing approach becomes immensely expensive due to the need for faster and higher capability processors along state-of-the-art computer hardware. Apart from parallel testing, systematic random testing [18] could also be another option. However, systematic random testing tends to dwell on unfair distribution of test cases.

A more recent and systematic solution to this problem is based on pairwise testing strategy. Here, any two combinations of parameter values are to be covered by at least one test [3, 17]. Because combinatorial explosion problem is NP-complete, it is often unlikely that efficient strategy exists that can always generate optimal test set (i.e. each interaction pair is covered by only one test). Furthermore, the size of the minimum pairwise test set also grows logarithmically with the number of parameter and

quadratically with the number of values [3]. Motivated by such a challenge, we have developed an efficient pairwise test data generation strategy, called G2Way. G2Way is our research vehicle to investigate the effectiveness of a pairwise strategy as far as software testing is concerned.

This paper is organized as follows. Section 2 highlights the related work. Section 3 describes the G2Way strategy is details. Section 4 highlights our evaluation as well as comparison against existing strategies in terms of the execution time as well as the number of generated test data. Finally, section 5 gives our conclusion.

2. Related Work

In order to practically address the combinatorial explosion problem discussed earlier, different pairwise strategies exist. According to Yu et al [16], existing strategies can be categorized into two categories based on the dominant approaches, that is, algebraic approaches or computational approaches.

Algebraic approaches construct test sets using pre-defined rules or mathematical function [16]. Thus, the computations involved in algebraic approaches are typically lightweight, and in some cases, algebraic approaches can produce the most optimal test sets. However, the applicability of algebraic approaches is often restricted to small configurations [14, 16]. Orthogonal arrays (OA) [8, 9] and covering arrays (CA) [8, 19] are typical example of the strategies based on algebraic approach. Some variations of the algebraic approach also exploit recursion in order to permit the construction of larger test sets from smaller ones (see reference [13]).

Unlike algebraic approaches, computational approaches often rely on the generation of the all pair combinations. Based on all pair combinations, the computational approaches iteratively search the combinations space to generate the required test case until all pairs have been covered. In this manner, computational approaches can ideally be applicable even in large system configuration. However, in the case where the number of pairs to be considered is significantly large, adopting computational approaches can be expensive due to the need to consider explicit enumeration from all the combination space.

Adopting the computational approaches as the main basis, an Automatic Efficient Test Generator (or AETG) [2, 3] and its variant (AETGm) [4], employs a greedy algorithm to construct the test case, that is, each test covers as many uncovered combinations as possible. Because AETG uses random search algorithm, the generated test case is highly non-deterministic (i.e. the same input parameter model may lead to different test suites [7]). Other variants to AETG that use stochastic greedy algorithms are: GA (Generic Algorithm) and ACA (Ant Colony Algorithm) [11]. In some cases, they give optimal solution than original AETG, although they share the common characteristic as far as being non-deterministic in nature.

In Parameter Order (IPO) strategy [17] builds a pairwise test set for the first two parameters. Then, IPO strategy extends the test set to cover the first three parameters, and continues to extend the test set until it builds a pairwise test set for all the parameters. In this manner, IPO generates the test case with greedy algorithms similar to AETG. Nevertheless, apart from deterministic in nature, covering one parameter at a time allows the IPO strategy to achieve a lower order of complexity than AETG.

Based on computational approach, Schroeder and Korel [10] developed a rather unique combinatorial strategy based on the input and output relationship. If one or more parameters are known to have insignificant effect on the system (i.e. don't care), then the strategy randomly selects the appropriate replacement of the don't care value in order to perform the reduction. Although useful for system with known input output relationship, no reduction is possible if all the parameters have the same importance.

A more recent strategies based on computational approaches are IRPS [15] and AllPairs [1]. Like IPO, IRPS is deterministic in nature. Unlike IPO and other computational strategies, IRPS focuses on efficient data structure for storing and searching pairs. In this manner, IRPS appears to be the only strategy that supports higher order interactions of parameters (i.e. from pairwise up to 13 ways).

Similar to IRPS and IPO, All Pairs strategy (i.e. downloadable tool) appears to share the same property as far as producing deterministic test cases is concerned although little is known about the actual strategies employed due to limited availability of references [1].

As far as other non-greedy strategies are concerned, some approaches opted to adopt heuristic search techniques such as hill climbing and simulated annealing (SA) [14]. Briefly, hill climbing and simulated annealing strategies start from some known test set. Then, a series of transformations were iteratively applied (starting from the known test set) to cover all the pairwise combinations [14]. Unlike AETG, IPO, IRPS and All Pairs strategy, which builds a test set from scratch, heuristic search techniques can predict the known test set in advanced. However, there is no guarantee that the test set produced are the most optimum.

3. The G2Way Strategy

A backtracking algorithms and search heuristics has been discussed in Jun and Jian [14]. Although useful, the work employed exhaustive search method typically requiring long execution time and may be restricted to small number of configuration. Although similar in name, G2Way is designed to be a flexible heuristic and does not rely on exhaustive search methods. In fact, G2Way relies on computational backtracking search procedure, which goes through the uncovered pairs through recombination as a way of getting the minimum test cases.

Adopting the computational approaches as its basis, the G2Way strategy actually depends on two algorithms: the pair generation algorithm and the backtracking algorithm.

• The Pair Generation Algorithm

The pair generation algorithm works as follows. Firstly, the algorithm finds the loop edge for the 2-way interaction (i.e. based on the number of defined parameters, *p*). Then, the algorithm performs index searches through a loop from 0 to 2^p -1. Here, for each index, the algorithm coverts the number to binary format. Now, if the number of binary one's in the index is equal to 2 (i.e. pairwise interaction), then that index is put in the index set.

As illustration, consider an example of a system having 3 parameters (P2, P1, P0), each of which has (1,3,2) values respectively. In this case, based on the number of parameters, the loop edge is 7 (i.e. 2^3 -1). The index searches loop found 3 indexes having two one's, that is (3,5,6) respectively (see Table 1).

Table 1. Index search

Index	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Going back to the pair generation algorithm, a row of possible pairwise values combination for each parameter can be now generated by recombining all the pair values for each parameter. Here, each index will contain a number of pairs (equals to the multiplication of values defined in each shared parameter). For our example, the first index will have 3x2 pairs, the second index will have 2x1 pairs, and third index will have 1x3 pairs. Hence, the total pairs are 11.

To ensure efficient implementation (i.e. reducing time and space requirements), the pair generation algorithm exploits row indexes to facilitate the storing and searching of pairs, the technique similar to IPOG [16]. Here, row indexes are used to store the indexes of the pairs, which in turns are a structure of bits. Using our example, row index 0 (corresponds to (P0,P1) pairs) stores 6 pairs which are indicated as bits b0 to b5. Similarly, row index 1 stores 2 pairs and row index 2 stores 3 pairs.

Table 2. Row index

Row Index	Index		b5	b4	b3	b2	b1	b0
0	3	→	1	1	1	1	1	1
1	5	→	0	0	0	0	1	1
2	6	→	0	0	0	1	1	1

```
Algorithm Pairs_Generation ( )
1: begin
2:    initalize S_p ={} where S_p represents the pair set
3:    let n_∑ = {n_0.....n_m} where n_∑ represents the values defined for each parameter, m = maximum no of parameters
4:    let p = {p_0 ..p_j}, where p represents the sorted set of sets of values defined for each parameter
5:    for index=0 to 2^m - 1
6:     begin
7:        let b = binary number
          b = convert index to binary
8:        if (the no of '1's in b = 2)
9:        begin
10:          calculate number of possible combinations (PC_i ) between the partial sets of values
11:            for the shared parameters
12:              begin
13:                multiply {n_x x ny} values from n_∑
14:                set the bits group (equal to PC_i ) in the index row to 1
15:             end
16:      end
17:    end
18:   return S_p
19: end
```

Figure 1. Pair generation algorithm

Based on the aforementioned discussion, the detail of the algorithm for pair generation is shown in Figure 1 given earlier.

- **The Backtracking Algorithm**

The backtracking algorithm iteratively traverses the pairwise sets in order to combine pairs with common parameter values in order to complete a test suite (hence, the algorithm is called *backtracking*). To ensure correct test set (i.e. each pair is covered at least once), pairs are combined if and only if the combination covers the most uncovered pairs. In the case where some pairs cannot be combined (i.e. due to the fact that the values are not uniform), the backtracking algorithm falls back to the first define values. In this manner, the pairs can still be covered. Finally, once, the pairs are covered, they are deleted from the pairwise sets. Hence, the algorithm ensures that all the pairs are covered when the pairwise set is empty.

Based on the above discussion and using the pair generation algorithm, the backtracking algorithm can be summarized in Figure 2.

4. Evaluation

Our evaluation has three main goals. The first goal is to demonstrate the correctness of the strategy as well as to assess whether or not the generated test cases are correct (i.e. each pair appears at least once). The second goal is to assess the effectiveness of the G2Way strategy for pairwise test data generation. Finally, the third goal is to compare the performance of G2Way against existing strategies particularly in terms of the size and the time taken to produce these test sets. In the next sub-sections, we will present our complete evaluations based on the aforementioned goals.

4.1 Demonstration of Correctness

To demonstrate the correctness of the G2Way strategy, we select a web-based configuration example

```
Algorithm Backtracking (Sp: Set)
1: begin
2:    initialize St ={}with empty set, where St represents the generated test cases set
3:    for the first two parameters
4:    begin
5:        create partial the test cases by selecting best values for higher parameters
          {P3....Pj}, that covers the maximum number of uncovered pairwise combinations in Sp.
6:        store generated test cases in St, and remove covered pairs from Sp (by set zero values to indicated bits).
7:    end
8:    while still found elements in Sp
9:    begin
10:       add a new element in the St set with empty fields.
11:       bring the first uncovered combination, decompose it to the initial value, fill it in the element set
12:       for 2nd uncovered combination
13:       begin
14:          decompose uncovered combination
15:          if (current pair element in Sp can be combined with other pair element)
16:          begin
17:             count number of uncovered combination
18:             if (has most uncovered pairs)
19:             begin
20:                fill it in the element set
21:             end
22:          end
23:       end
24:       if (the element set does not have matching pair)
25:       begin
26.          select the first element as default values to missing parameter
27.       end
28.       store it in St and remove the covered pairs from Sp
29:    end
30:   return St
31: end
```

Figure 2. Backtracking algorithm

as a case study. The rationale for using this example stemmed from the fact that historically the same data inputs have been used by other researchers in the area (e.g. in [5]). By adopting the same data inputs, objective comparison may be made amongst different strategy implementation.

Overall, the web-based configuration example consists of 4 parameters, each of which has 3 values as seen in Table 3.

Table 3. Web based system

P1	P2	P3	P4
Netscape	Windows	LAN	Local
IE	Macintosh	PPP	Networked
Firefox	Linux	ISDN	Screen

Based on the web-based configuration example above, the following test set has been generated using G2Way (see Table 4). Here, G2Way produces 10 test data.

Table 4. Suggested test set

T#	P1	P2	P3	P4
1	Netscape	Windows	LAN	Local
2	IE	Windows	PPP	Networked
3	Firefox	Windows	ISDN	Screen
4	Netscape	Macintosh	PPP	Screen
5	IE	Macintosh	LAN	Local
6	Firefox	Macintosh	LAN	Networked
7	Netscape	Linux	ISDN	Networked
8	IE	Linux	LAN	Screen
9	Firefox	Linux	PPP	Local
10	IE	Macintosh	ISDN	Local

In order to investigate whether or not the all pairs are covered, it is necessary to tabulate all the pairs. In this case, the pairwise interactions of parameters are between (P1,P2), (P1,P3), (P1,P4), (P2,P3), (P2,P4) and (P3,P4). Based on these interactions, the expected total pairs will be 54 (i.e. 9 pairs/interactions x 6 interactions).

As discussed earlier, we will focus on demonstrating the correctness of the G2Way strategy by analyzing the resulting test case set. Here, we aim to show that G2Way gives optimum results, that is, all pairs of combinations are covered at least once. Table 5 lists all the pairs along with the test cases generated by G2Way strategy that cover them (denoted as **T#**). Referring to Table 5, we observe that each combination pair appears at least once (which means that the generated test cases include all generated pairs) and there is no missing pair (which means that our strategy is correct).

Table 5. Pairwise coverage

Pair Combination	T#	Pair Combination	T#
Netscape, Windows	1	IE, Windows	2
Netscape, LAN	1	IE, LAN	5
Netscape, Local	1	IE, Local	5
Netscape, Macintosh	4	IE, Macintosh	5
Netscape, PPP	4	IE, PPP	2
Netscape, Networked	7	IE, Networked	2
Netscape, Linux	7	IE, Linux	8
Netscape, ISDN	7	IE, ISDN	10
Netscape, Screen	4	IE, Screen	8
Windows, LAN	1	Macintosh, LAN	5
Windows, Local	1	Macintosh, Local	5
Windows, PPP	2	Macintosh, PPP	4
Windows, Networked	2	Macintosh, Networked	6
Windows, ISDN	3	Macintosh, ISDN	10
Windows, Screen	3	Macintosh, Screen	4
LAN, Local	1	PPP, Local	9
LAN, Networked	6	PPP, Networked	2
LAN, Screen	8	PPP, Screen	4
Linux, LAN	8	Firefox, Windows	3
Linux, Local	9	Firefox, LAN	6
Linux, PPP	9	Firefox, Local	9
Linux, Networked	7	Firefox, Macintosh	6
Linux, ISDN	7	Firefox, PPP	9
Linux, Screen	8	Firefox, Networked	6
ISDN, Local	10	Firefox, Linux	9
ISDN, Networked	7	Firefox, ISDN	3
ISDN, Screen	3	Firefox, Screen	3

4.2 Effectiveness of G2Way Strategy

To demonstrate the effectiveness of the G2Way strategy for pairwise test data generation, the FileChooserDemo program [12] has been chosen as independent open source code (i.e. downloadable from the SUN Microsystem website). As the name suggests, the FileChooserDemo is a program to demonstrate various Java GUI for selection based controls (see Figure 3).

Referring to Figure 3, the FileChooserDemo program has 14 parameters (1 4 valued parameters, 2 3 valued parameters, 11 2 valued parameters), the parameters in details are:

P1= Look and Feel (Metal, CDE/Motif, Windows, Windows Classic)
P2= Dialog Type (Open, Save, Custom)
P3= File and Directory Options (Just Select Files, Just Select Directories, Select Files or Directories),
P4= Show “All Files” Filter (Checked, Not),
P5= Show JPG and GIF Filters (Checked, Not),
P6= With File Extensions (Checked, Not),

P7= Show Hidden Files (Checked, Not),
P8= Use FileView (Checked, Not),
P9 = Use Preview (Checked, Not),
P10= Embed in Wizard (Checked, Not),
P11= Show Control Buttons (Checked, Not),
P12= Enable Dragging (Checked, Not),
P13= File and Directory Options (Single Selection, Multi Selection)
P14=Show File Chooser (Select, Cancel).

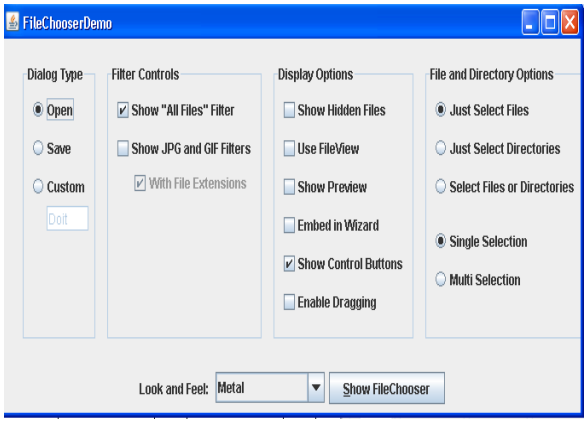

Figure 3. FileChooserDemo interface

Based on the number of parameters, considering all exhaustive combinations would require $4^1x3^2x2^{11}$ = 73728 test cases. Considering pairwise testing and using G2Way strategy, the test cases are reduced to merely 15 (see Table 6).

Here, we are interested to investigate whether or not the 15 suggested test cases are sufficient to test FileChooserDemo program whilst giving acceptable coverage (i.e. in terms of the program areas, blocks or paths exercised by the test data). In the absence of the specification, we believe, it is sufficient to evaluate our test execution based on whether or not the program behaves as expected.

To help measure coverage, we have adopted EMMA [6], an open source test coverage tool from SourceForge. Using EMMA, a number of coverage metrics can be reported. The first coverage metric is the class coverage. In EMMA, the class coverage refers to the ratio of the covered classes over the total number of classes. The second metric is the method coverage. Here, the method coverage refers to the ratio of the covered methods over the total number of methods. The third metric is the block coverage, defined as the total covered blocks over the total blocks. Finally, the last metric is the line coverage, defined as the covered lines over the total number of lines.

Executing the 15 suggested test cases, we observe no errors as the program behaves as expected. Using EMMA, we obtain the following coverage results (see Table 7). Noted here is the fact that these metrics are calculated based on the FileChooserDemo implementation consisting of 9 classes, 42 methods, 2136 blocks, and 450 lines.

Table 7. Percentage coverage

Class Coverage	Method Coverage	Block Coverage	Line Coverage
100%	83%	96%	94%

Referring to the coverage results tabulated in Table 7, it is evident that the pairwise test data set generated by G2Way is reasonably effective to exercise various coverage metrics (i.e. 100% of class coverage, 83% of method coverage, 96% of block coverage and 94% of line coverage). In fact, a closer look to the source code reveals that uncovered code comes from the exception handling mechanism as well as dead code (which can not be detected even with exhaustive combinations). Thus, we conclude that G2Way strategy is effective for pairwise test data generation.

Table 6. Suggested test suite

T#	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
1	Metal	Open	J.S.F	T	T	T	T	T	T	T	T	T	Single	Select
2	CDE/Motif	Open	J.S.D	F	F	F	F	F	F	F	F	F	Multi	Cancel
3	Windows	Open	F or D	T	F	T	F	T	F	T	F	T	Multi	Select
4	Win.Classic	Open	J.S.F	F	T	F	T	F	T	F	T	F	Single	Cancel
5	Metal	Save	J.S.D	T	T	F	F	T	T	F	F	T	Single	Cancel
6	CDE/Motif	Save	J.S.F	T	F	T	T	F	F	T	T	F	Multi	Select
7	Windows	Save	J.S.F	F	T	T	T	T	T	F	F	F	Multi	Select
8	Win.Classic	Save	F or D	T	T	T	T	T	F	T	T	T	Single	Cancel
9	Metal	Custom	F or D	F	F	F	T	F	T	T	T	T	Single	Select
10	CDE/Motif	Custom	J.S.F	T	T	T	F	T	T	F	T	T	Single	Cancel
11	Windows	Custom	J.S.D	T	T	T	T	F	F	T	T	F	Single	Select
12	Win.Classic	Custom	J.S.D	T	F	T	F	T	T	T	F	T	Multi	Select
13	CDE/Motif	Open	F or D	T	T	T	T	T	T	F	T	F	Single	Select
14	Windows	Open	J.S.F	T	T	F	T	T	T	T	T	T	Single	Cancel
15	Metal	Open	J.S.F	T	T	T	T	T	F	T	T	F	Multi	Select

4.3 Comparison with other strategies

Concerning comparison, we have identified the following existing strategies that support pair wise testing: AETG [2, 3] , AETGm [4], IPO [17], SA [14], GA [11] , ACA [11], and AllPairs tool [1]. We consider eight system configurations.

S1: 3 3-valued parameters
S2: 4 3-valued parameters,
S3: 13 3-valued parameters,
S4: 10 10-valued parameters,
S5: 10 15-valued parameters,
S6: 20 10-valued parameters,
S7: 10 5-valued parameters
S8: 1 5-valued parameters, 8 3-valued parameters and 2 2-valued parameters.

Table 8 shows the size of the test set generated by each strategy, and Table 9 shows the execution time for each system. All the problem instances and data for the existing strategies are taken from [15], except for All Pairs tool (which is free for download, hence, we can run it in our platform). Entries marked with NA are data that are not available in these papers.

In order to ensure objective comparison, we summarize the hardware and software platform used.

- AETG, AETGm, SA: Intel P IV 1.8 Ghz, C++ programming language, Linux Operating System
- IPO: Intel P II 450 Mhz, Java programming language, Windows 98 operating system
- CA, ACA: Intel P IV 2.26 GhZ, C programming language, Windows XP operating system
- All Pairs: Intel P IV 1.8 Ghz, 512 MB RAM, Perl programming language, and Windows Vista operating system
- G2Way: Intel P IV 1.8 Ghz, 512 MB RAM, C++ programming language, Windows Vista operating system.

Referring to Table 8, G2Way and All pairs generates the same number of test cases for S1. For S2, AETG, IPO, SA, GA, and ACA outperforms G2Way and All Pairs. For S3, AETG gives the best result as compared to all other strategies. For S4, G2Way comes second to ACA. For S5, G2Way outperforms IPO and Allpairs (i.e. no data is available for other strategies). For S6, AETG outperforms all other strategies. For S7, G2Way outperforms other strategies. Finally, for S8, GA and SA yield the best result.

From the above given results, it can be seen that no strategies can claim dominance over the others. Although having a lot of entries with NA, AETG appears to give the best overall results. IPO gives good result with small configuration, but appears to generate more test set with high configuration. Perhaps, All pairs can be comparable to G2Way as it gives similar no of test set for small configuration. However, G2Way often gives better results for high configuration as compared to All pairs.

Concerning execution, it must be stressed that no fair comparison can be made in terms of execution time due to the differences in the computing environment as well as the unavailability of the open source code or executable code to run in our platform. As noted earlier, we only manage to get access to All pairs to run in our platform. As a general observation, however, we believe the execution time for G2Way is acceptable as compared with other strategies (see Table 9). Notwithstanding the differences in the computing environment, it is clear that IPO outperformed other

Table 8 – Comparison in terms of the size of the generated test set

System	AETG	AETGm	IPO	SA	GA	ACA	ALL Pairs	G2Way
S1	NA	NA	NA	NA	NA	NA	10	10
S2	9	11	9	9	9	9	10	10
S3	15	17	17	16	17	17	22	19
S4	NA	NA	169	NA	157	159	177	160
S5	NA	NA	361	NA	NA	NA	390	343
S6	180	198	212	183	227	225	230	200
S7	NA	NA	47	NA	NA	NA	49	46
S8	19	20	NA	15	15	16	21	23

Table 9 – Comparison in terms of the execution time (in seconds)

System	AETG	AETGm	IPO	SA	GA	ACA	All Pairs	G2Way
S1	NA	NA	NA	NA	NA	NA	0.08	0.047
S2	NA	NA	NA	NA	NA	NA	0.23	0.062
S3	NA	NA	NA	NA	NA	NA	0.45	0.25
S4	NA	NA	0.3	NA	866	1180	5.03	2.906
S5	NA	NA	0.72	NA	NA	NA	10.36	7.438
S6	NA	6,001	NA	10,833	6,365	7,083	23.3	1,753
S7	NA	NA	0.05	NA	NA	NA	1.02	0.687
S8	NA	58	NA	214	22	31	0.35	0.33

strategies as far as execution time is concerned. This may be due to fact that IPO is deterministic algorithm and need only one run. For these reason, it requires much less time to execute than others. Although giving the best overall results in terms of the number of generated test set, the execution time for AETG is unknown.

5. Conclusion

In this paper, we propose a novel deterministic computational strategy for pairwise testing, called G2Way. Comparing to other strategies, our initial evaluation results are encouraging with acceptable test size and execution time. As part as our future work, we are currently investigating a more general strategy capable of handling more than 2 way interactions.

Acknowledgement

This research is funded by the USM Fundamental Grants – Investigating Heuristic Algorithm to Address Combinatorial Explosion Problem for Hardware and Software Testing.

References

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

[2] 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*, 23(7), July 1997, pp. 437-444.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, G. C. Patton, and N.J. Bellcore, "The Combinatorial Design Approach to Automatic Test Generation", vol. 13: *IEEE Software*, Sep 1996, pp. 83-89.

[4] M.B. Cohen, "*Designing Test Suites For Software Interaction Testing*", School of *Computer Science, Univ. of* Auckland, PhD Thesis (2004).

[5] C. J. Colbourn, M.B. Cohen, and R.C. Turban, "A Deterministic Density Algorithm for Pairwise Interaction Coverage", *In Proc. of the IASTED Intl. Conference on Software Engineering*, February 2004, pp. 242-252.

[6] "EMMA: a free Java code coverage tool", Available from: http://emma.sourceforge.net/, 2006.

[7] M. Grindal, J. Offutt, and S.F. Andler, "Combination Testing Strategies: A Survey", *GMU Technical Report ISE-TR-04-05*, July 2004.

[8] A. Hartman and L. Raskin. "Problems and Algorithms for Covering Arrays", Discrete Math., July 2004, pp. 149-156, 2004, Elsevier.

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

[10] P. J. Schroeder and B. Korel, "Black-Box Test Reduction Using Input-Output Analysis", in *Proc. Of the International Symposium on Software Testing and Analysis (ISSTA 2000)* Portland, OR, USA, 2000.

[11] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing", i*n Proc. of the 28th Annual International Computer Software and Applications Conference COMPSAC 2004*, Hong Kong, 2004, pp. 72-77.

[12] SUN. "How to Use File Choosers", Available from: http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html.

[13] A.W. Williams and R.L. Probert, "A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces", in *Proc. of the 7th International Symposium on Software Reliability Engineering*, 1996, pp. 246-254.

[14] J. Yan and J. Zhang, "Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing", in *Proc. of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06).* vol. 1, 2006, pp. 385-394.

[15] M.I. Younis, K.Z. Zamli, and N.A.M. Isa, "IRPS –An Efficient Test Data Generation Strategy for Pairwise Testing,", in *Proc. of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems KES2008* Zagreb, Croatia, 2008.

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

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

[18] K.Z. Zamli, N.A. M. Isa, M.F.J. Klaib, and S.N. Azizan, "Designing a Combinatorial Java Unit Testing Tool", in *Proc. of the IASTED Intl. Conference on Advances in Computer Science and Technology (ACST 2007),* Phuket, Thailand, April 2007.

[19] L. Zekaoui, "*Mixed Covering Arrays on Graphs And Tabu Search Algorithms*", *Ottawa-Carleton Institute for Computer Science*, University of Ottawa, Canada, Master Thesis (2006).