G2WAY: A PAIRWISE TEST DATA GENERATION STRATEGY WITH AUTOMATED EXECUTION SUPPORT

Kamal Zuhairi Zamli[1], Mohammed Fadel Jamil Klaib, Mohammad Issam Younis and Ong Hui Yeh

School of Electrical and Electronic Engineering
Universiti Sains Malaysia Engineering Campus

eekamal@eng.usm.my[1]

ABSTRACT

This paper discusses a new strategy, called G2Way, for pairwise test data generation. Unlike existing strategies, G2Way also supports (concurrent) automated execution integrated within the strategy itself. Furthermore, empirical evidence demonstrates that G2Way, in some cases, outperformed existing strategies in terms of the number of generated test data (as compared to that of AETG and its variations, IPO, SA, GA, ACA, and All Pairs). Notwithstanding the differences in the computing environment employed as well as the overhead incurred to permit automated execution, the test generation time is also within reasonable value.

Keywords: Way, Combinatorial Testing, Pairwise Testing.

INTRODUCTION

Modern society in today's digital era depends heavily on computer software in almost every aspect of daily life. 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 dependency 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 easily be changed and customized as the need arises.

Our dependencies on software often raise many issues as far as reliability is concerned. Faulty software can cause severe data loss, crash the computer and do other unexpected incidents that would squander the resource assets.

For this reason, testing becomes immensely important. In order to ensure an acceptable level of quality and reliability of a typical software product, it is desirable to test every possible combination (Copeland, 2004) of input data under various configurations (e.g. by also considering the running software and hardware environments). Due to the combinatorial explosion problem, consideration of all exhaustive testing is impossible. Resource constraints, costing factors as well as strict time- to-market deadlines are amongst the main factors that inhibit such consideration. Earlier work suggests that pairwise sampling strategy (i.e. based on two-way parameter interaction) can be effective (Klaib, Zamli, Isa, Younis & Abdullah, 2008). In fact, many helpful pairwise sampling strategies have been developed in the literature.

Much useful advancement has been achieved in the last 10 years particularly to facilitate the test planning process, that is, in terms of systematically minimizing the test data to be considered for testing (i.e. based on the pairwise parameter interactions). Despite such a significant progress, the integration and automation of the strategies from the planning process to execution appears to be lacking. In current practice, the sampled test data need to be manually extracted and converted to some acceptable format before they can be executed (e.g. by a human tester, a code driver or a third party execution tool). This lack of integration and automation between test planning and execution can potentially burden the test engineers especially if the module to be tested is significantly large.

Apart from the integration and automation issues, strategizing to sample and construcing minimum test set from the exhaustive test space is also a NP complete problem (Tai & Lei, 2002; Shiba, Tsuchiya & Kikuno, 2004; Lei & Tai, 1998). As such, it is often unlikely that an efficient strategy exists that can always generate an optimal test set (i.e. each interaction pair is covered by only one test). Motivated by such challenges, we are currently investigating a new strategy for pairwise testing, called G2Way. The G2Way strategy, unlike other strategies, aims to automate and integrate test planning and execution as well as support an efficient generation of pairwise test data. This paper describes the G2Way strategy (Klaib et al., 2008) as well as compares its effectiveness against existing strategies including AETG (Cohen, Dalal, Fredman & Patton, 1997) and AETGm (Cohen, Gibbons, Mugridge & Coulbourn 2003), IPO (Lei & Tai, 1998), SA (Yan & Zhang, 2006), GA (Shiba et al., 2004), ACA (Shiba et al., 2004), and All Pairs (Bach, 2009). Empirical evidence demonstrates that G2Way, in some cases, outperformed other strategies in terms of the number of generated test data. Additionally, notwithstanding the differences in the computing environment employed as well as the overhead incurred to permit integration between test planning and execution, the test generation time is also within reasonable time.

RESEARCH METHODOLOGY

The research methodology is divided into three parts as follows:

1. Survey of the state-of-the-art: A survey of the state-of-the-art regarding the pairwise test data implementation is conducted in order to find the suitable implementation methods. Based on the survey, the implemented strategy, G2Way, will be developed in an effort to enhance the existing strategies with automation and execution support.
2. Design and Implementation: Here, G2Way will be designed, implemented, and finally tested for correctness.
3. Evaluation: In order to validate the integration of the test planning and test execution, case study evaluations are conducted to showcase the effectiveness of the G2Way strategy as far as the art-of-the-practice is concerned.

Related Work

As discussed earlier, the focus of the current pairwise strategies has been on test planning. As such, to the best of our knowledge, no significant work has been reported on integrating and automating the pairwise strategies for both test planning and execution.

Considering the approaches adopted by the existing strategies, they can be categorized into two categories (Lei, Kacker, Kuhn, Okun & Lawrence, 2007), that is, algebraic approaches and computational approaches.

Algebraic approaches construct test sets using pre-defined rules or mathematical functions (Lei et al., 2007). 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 (Lei et al., 2007), (Yan & Zhang, 2006). Orthogonal arrays (OA) (Hartman & Raskin, July 2004; Hedayat, Sloane & Stufken, 1999) and covering arrays (CA) are typical examples 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 (Williams & Probert, 1996).

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 (Cohen et al., 1997; Cohen, Dalal, Kajla & Patton, 1994), and its variant (AETGm) (Cohen et al., 2003), employ 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 (Grindal et al., 2005)). Other variants to AETG that use stochastic greedy algorithms are: GA (Generic Algorithm) and ACA (Ant Colony Algorithm) (Shiba et al., 2004). In some cases, they give optimal solutions than the original AETG although they share the common characteristic as far as being non-deterministic in nature.

In Parameter Order (IPO) strategy (Lei & Tai, 1998) 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 being 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 (2000) developed a rather unique combinatorial strategy based on the input and output relationship. If one or more parameters are known to have an 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 systems with known input-output relationships, no reduction is possible if all the parameters have the same importance.

In more recent strategies based on computational approaches are IRPS (Younis, Zamli & Isa, 2008) and All Pairs (Bach, 2009). 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 is capable of supporting higher order interactions of parameters.

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 (Bach, 2009).

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) (Yan & Zhang, 2006). 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 (Yan & Zhang, 2006). Unlike AETG, IPO, IRPS and All Pairs strategy, which build a test set from scratch, heuristic search techniques can predict the known test set in advance. However, there is no guarantee that the test sets produced are the most optimum.

The G2Way Strategy

The overall view of the G2Way strategy (Klaib et al., 2008) can be seen in Figure 1. Here, the base test data is first specified using a special markup language.

Tthe markup language is based on our earlier work described in (Zamli et al., 2007). Figure 2 illustrates a snapshot of a specification of the base input test data expressed using the markup language (i.e. keywords are shown in bold). Apart from capturing the input test data, the markup language also allows the definition of the values, data types, and access scopes as well as the methods/functions that need to be tested. As will be seen later, it is this information that will be used by the executor algorithm to execute the test data, that is, by automatically generating a code driver to automate the actual testing process.

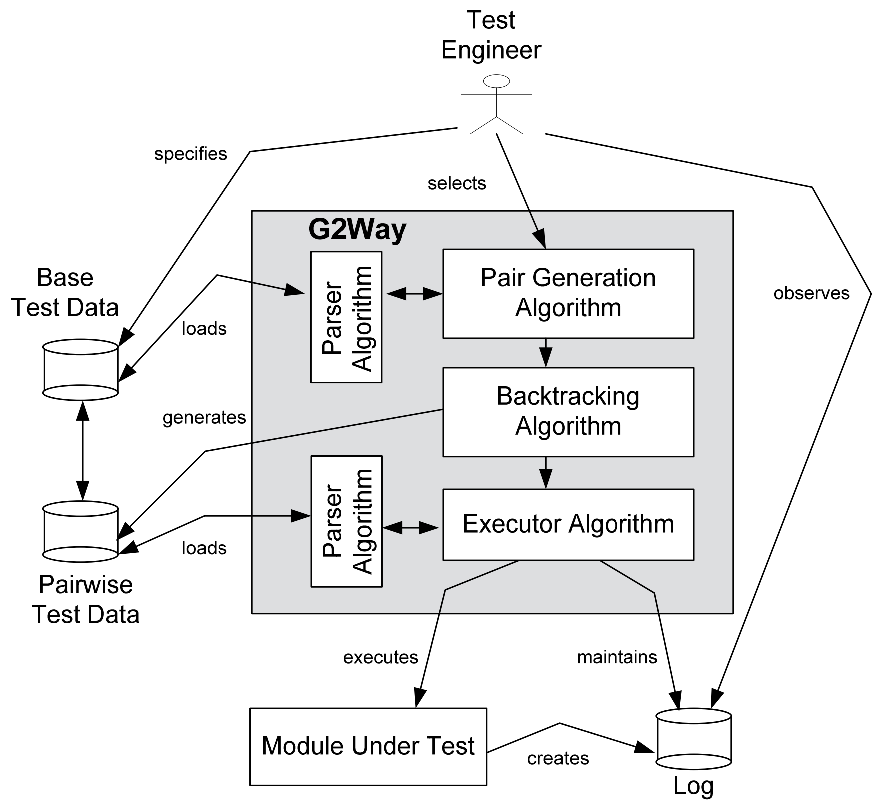


Figure 1. Overview of the G2Way Strategy

Then, upon the execution of the G2Way strategy, the parser algorithm will load the parameter and values to be used by the pair generation algorithm (discussed later). Exploiting the result generated by the pair generation algorithm, the backtracking algorithm generates the pairwise test sets. Finally, upon completion, the backtracking algorithm forwards the results to the executor algorithm for execution. Noted here is the fact that the executor algorithm can also independently load the pairwise test sets for execution through the parser algorithm.

Having given a high level picture on how the G2Way strategy works, the next section highlights all the algorithms involved.

```
@FaultFile
////////////////////////////////////////
     Common Header Definition
////////////////////////////////////////
className : CollAccept
methodName : testAcceptance
specifier: public
paramTypes : 5
returnType: void
parameter : partypes[0]=Double.TYPE
parameter : partypes[1]=Double.TYPE
parameter : partypes[2]=Double.TYPE
parameter : partypes[3]=Double.TYPE
parameter : partypes[4]=Double.TYPE
////////////////////////////////////////
     Body - Test case 0
////////////////////////////////////////
arglist:arglist[0]=new Double(49)
arglist:arglist[1]=new Double(49)
arglist:arglist[2]=new Double(49)
arglist:arglist[3]=new Double(49)
arglist:arglist[4]=new Double(49)
////////////////////////////////////////
     Body - Test case 1
////////////////////////////////////////
arglist:arglist[0]=new Double(74)
arglist:arglist[1]=new Double(74)
 ……………
```

Figure 2. Snapshot of the Test Data Specification

(i) The Parser Algorithm

As the name suggests, the parser algorithm (see Figure 3) parses the module under test (specified in the fault file) to capture the necessary keywords and values to be used for pairwise generation and execution (e.g. the className, methodName, paramNo, paramTypes, and return type). Additionally, the parser algorithm also loads the parameters and values into the parameter and value sets.

```
Algorithm Parser (fault_file:File)
1: begin
2:   load fault_file
3:    let p = {} as empty set, where p represents the defined parameters
4:    let $n_\Sigma$ = {} as empty set, where $n_\Sigma$ represents the parameter values
5:   while not EOF (fault_file)
6:     begin
7:      read value
8:     if keyword in value = {className or methodName or paramNo or paramType or returnType or parameter}
9:        begin
9:         parse value
11:          assign value to className or methodName or paramNo or paramType or returnType or parameter
12:       end
10:  if keyword in value = {arglist[i]...arglist[paramNo]} where i<=0<=paramNo
11:        begin
12:          parse value
13:          assign i to the p set
14:          assign value to $n_\Sigma$ set
15:        end
16:    end
17: end
```

Figure 3. Parser Algorithm

(ii) 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 converts the number to binary format. Now, if the number of binary ones in the index is equal to 2 (i.e. pairwise interaction), then that index is put in the index set. As an 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 ones, 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

Table 2

Row Index

Row	**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

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. (see Table 2) 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 the 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, a technique similar to IPOG (Lei et al., 2007). Here, row indexes are used to store the indexes of the pairs, which in turn 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. Based on the aforementioned discussion, the detail of the algorithm for pair generation is summarized in Figure 4.

```
Algorithm Pair_Generation ( )
1: begin
2:   let Sp ={} as empty set, where Sp  represents the
      pair set
3:    let n∑ =  {n0......nm} where n∑ represents the
      values defined for each parameter, m = max no of
       parameters
4:   let p = {p0 ..pj}, 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
            (PCi )between the partial sets of values
11:          for the shared parameters
12:             begin
13:               multiply {nx x ny} values from n∑
14:               set the bits group (equal to PCi) in
                  the  index row to 1
15:             end
16:        end
17:    end
18: end
```

Figure 4. Pair Generation Algorithm

(iii) 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 defined 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 5.

```
Algorithm Backtracking (Sp: Set)
1: begin
2:   let St ={} as 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
7:   remove covered pairs from Sp(by setting zero values
     to indicated bits).
8:    end
9:   while still found elements in Sp
10:   begin
11:    add a new element in the St set with empty fields
12:     bring the first uncovered combination, decompose
          and fills the initial value in the element set
13:      for 2nd uncovered combination
14:        begin
15:          decompose uncovered combination
16:          if (current pair element in Sp can be
               combined with other pair element)
17:            begin
18:              count number of uncovered combination
19:              if (has most uncovered pairs)
20:               fill it in the element set
21:            end
22:     if (the element set does not have matching pair)
23:       select the first element as default values to
            missing parameter
24:   store it in St and remove the covered pairs from Sp
25:    end
26:   end
27:  return St
28: end
```

Figure 5. Backtracking Algorithm

(iv) Executor Algorithm

As discussed earlier, G2Way can serve both as a pairwise test planning strategy and as a test execution and automation tool. In this case, only when real data values are specified in the test data specification (i.e. in the fault file) can G2Way support automated execution through its executor algorithm. Here, the executor algorithm simply takes the name of the defined class, methods, as well as its associated parameters and values, and automatically generate a test driver to drive execution. In this manner, concurrent execution is possible through the judicious use of threads. The complete description of the executor algorithm is depicted in Figure 6.

```
Algorithm Executor(className,methodName,paramNo,param
Types,returnType,parameter:String; St:Set)

1: begin
2:  for each test case, i, in St set
3:   begin
4:    create a public driver class
5:   generate and compile the main method for the driver
class with specific call to the method under test
      by passing the ith test case from St
6:    instantiate a driver object
7:    if (thread<limit)
8:       spawn and execute the thread for the created
driver object
9:    capture the result and errors in log, if any
10:  end
11: end
```

Figure 6. Executor Algorithm

Evaluation

Our evaluation has four 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. The third goal is to demonstrate the applicability of G2Way for test planning and execution. Finally, the fourth 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.

(i) Demonstration of Correctness

To demonstrate the correctness of the G2Way strategy, we select a web-based configuration example 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 [Colbourn, Cohen & Turban, 2004]). By adopting the same data inputs, objective comparisons 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. Here, G2Way produces 10 test data (see Table 4).

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 all the 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 analysing the resulting test case set. Here, we aim to show that G2Way gives efficient 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 the G2Way strategy that covers them (denoted as T#).

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

(continued)

Table 5

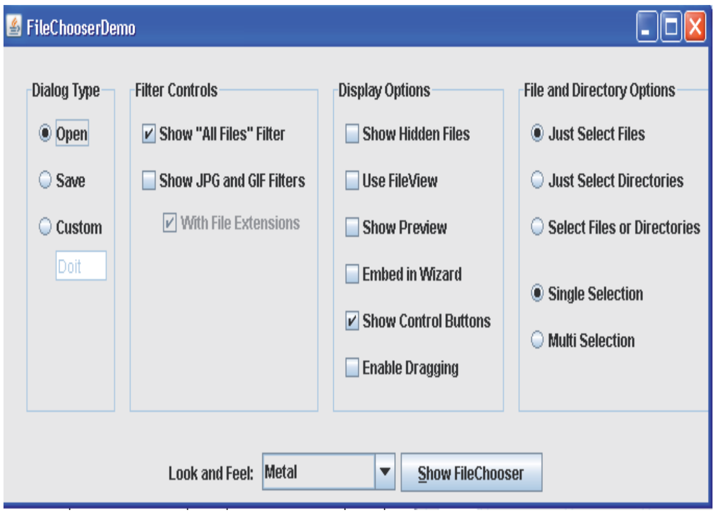
Pairwise Coverage

Pair Combination	T#	Pair Combination	T#
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

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).

(ii) Effectiveness of the G2Way Strategy

To demonstrate the effectiveness of the G2Way strategy for pairwise test data generation, the FileChooserDemo programme (SUN) has been chosen as an independent open source code (downloadable from the SUN Microsystem website). As the name suggests, the FileChooserDemo is a programme to demonstrate various Java GUI for selection based controls (see Figure 7).



Dialog Type: ● Open ○ Save ○ Custom

Filter Controls: ✓ Show "All Files" Filter; ☐ Show JPG and GIF Filters; ✓ With File Extensions

Display Options: ☐ Show Hidden Files; ☐ Use FileView; ☐ Show Preview; ☐ Embed in Wizard; ✓ Show Control Buttons; ☐ Enable Dragging

File and Directory Options: ● Just Select Files ○ Just Select Directories ○ Select Files or Directories; ● Single Selection ○ Multi Selection

Look and Feel: Metal

Figure 7. FileChooserDemo Interface

Referring to Figure 7, the FileChooserDemo programme has 14 parameters (1 4 valued parameters, 2 3 valued parameters, 11 2 valued parameters). The parameters in detail 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).

Based on the number of parameters, considering all exhaustive combinations would require $4^1 x 3^2 x 2^{11} = 73728$ test cases. Considering pairwise testing and using the G2Way strategy, the test cases are reduced to merely 15 (see Table 6). Here, we are interested in investigating whether or not the 15 suggested test cases are sufficient to test the FileChooserDemo programme whilst giving acceptable coverage (i.e. in terms of the programme 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 programme behaves as expected.

To help measure coverage, we have adopted EMMA (2006), 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.

Table 6

Suggested Test Set

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

Executing the 15 suggested test cases, we observe no errors as the programme 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, two conclusions can be made here. Firstly, 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). Secondly, in this programme, there is not much interaction among all the control interface parameters with each other. As will be seen later, interactions between parameters can play a significant role as far as coverage is concerned.

(iii) Applicability of the G2Way Strategy for Test Planning and Execution

In this section, we aim to demonstrate the applicability of G2Way for both test data generation and execution. Here, we opt to use the programme source codes which consist of highly interacting input variables (as will be discussed later). To do so, we hypothetically envisage a programme (called *college_acceptance*) that can automatically advise student's acceptance for college admission. In this programme, it is assumed that the college has four main departments, that is, Department of Mathematics, Department of Physics, Department of Biology, and Department of Computing. The acceptance criteria to any of the departments depends on the student's grade in high school for five subjects namely English, Mathematics, Physics, Biology, and Computer Science. In this hypothetical problem, the student can be accepted in one of the departments, only if:

(a) he/she passes all five subjects (i.e. each subject has a score of 50 % or better).
(b) he/she scores 75% or better in the related subject to the department he/she is applying for.
(c) the acceptance will be conditional if the English subject score is less than 75%.

Unlike the earlier assessment (i.e. the GUI-based FileChooser demo) which lacks parameter interaction, the acceptance criteria discussed here appears to be highly intertwined and interdependent with each other. Thus, it is expected that pairwise interaction may not be sufficient for a good coverage.

To serve as our case study, we have implemented the *college_acceptance* programme as a Java programme. The *college_acceptance* programme consists of 1 class, 2 methods, 594 blocks and 61 lines. The two methods in the programme are the main ( ) and the *testAcceptance* ( ) method. Here, the *testAcceptance* ( ) method takes five parameters of type double, for each of the subject scores (e.g. English, Mathematics, Physics, Biology, and Computer Science). Considering the subject scores with the criteria discussed earlier, a decision will be taken and printed as Incorrect Grades, Not Accepted in any department, Conditionally Accepted in specific department, or Accepted in specific department.

Using the equivalence partitioning technique, the grade level can be divided into three intervals. The first interval is between [0, 50]. The second interval is between [50, 75], and the final interval is between [75,100]. Here, the base test cases (see Table 8) can be selected in each of the intervals to cover all the valid values. In this case, the first value 49 belongs to the 1st interval. The second value 74 belongs to the second interval. Finally, the third value 76 belongs to the last interval. As an illustration, an excerpt snapshot of the test data specification for the base test data can be seen in Figure 2 given earlier.

Table 8

Base Test Cases

Maths	Physics	Biology	English	Computer Science
49	49	49	49	49
74	74	74	74	74
76	76	76	76	76

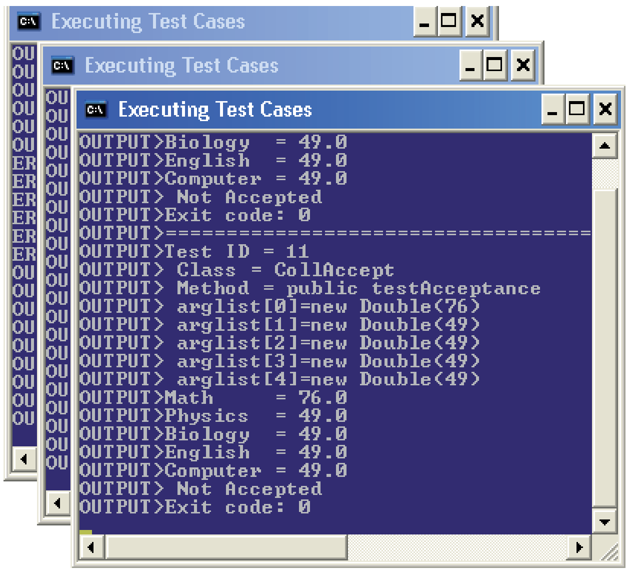
Based on the number of parameters, considering all exhaustive combinations would require $3^5 = 243$ test cases. Considering pairwise testing and using the G2Way strategy, the test cases are reduced to merely 14 (see Table 9).

Table 9

Suggested Test Set

T#	Math	Physics	Biology	English	Computer Science
0	49	49	49	49	49
1	74	49	74	74	74
2	76	49	76	76	76
3	49	74	74	76	49
4	74	74	49	49	76
5	76	74	49	74	74
6	49	76	76	74	49
7	74	76	49	76	74
8	76	76	74	49	76
9	74	74	76	49	74
10	74	49	49	49	49
11	76	49	49	49	49
12	49	49	49	49	74
13	49	49	49	74	76

Concurrently executing the 14 suggested test cases (see snapshot in Figure 8) using G2Way, we observe no errors as the programme behaves as expected.



```
OUTPUT>Biology   = 49.0
OUTPUT>English   = 49.0
OUTPUT>Computer = 49.0
OUTPUT> Not Accepted
OUTPUT>Exit code: 0
OUTPUT>================================
OUTPUT>Test ID = 11
OUTPUT> Class = CollAccept
OUTPUT> Method = public testAcceptance
OUTPUT> arglist[0]=new Double(76)
OUTPUT> arglist[1]=new Double(49)
OUTPUT> arglist[2]=new Double(49)
OUTPUT> arglist[3]=new Double(49)
OUTPUT> arglist[4]=new Double(49)
OUTPUT>Math      = 76.0
OUTPUT>Physics   = 49.0
OUTPUT>Biology   = 49.0
OUTPUT>English   = 49.0
OUTPUT>Computer = 49.0
OUTPUT> Not Accepted
OUTPUT>Exit code: 0
```

Figure 8. Concurrent Execution Snapshot

Again, using EMMA tool (2006), we obtain the following coverage results (see Table 10).

Table 10

Percentage Coverage

Class Coverage	**Method Coverage**	**Block Coverage**	**Line Coverage**
100%	100%	21%	16%

Theere are two conclusions that can be elaborated here. The first conclusion is that G2Way can support automated (concurrent) execution apart from its test generation capability.

The second conclusion is a more subtle one. As expected, referring to the coverage results tabulated in Table 10, the pairwise test data generated is not sufficiently enough to give a good coverage. In fact, for this highly interacting parameters implementation, there is a need to go for a higher order interaction in order to get a good coverage.

(iv) Comparison with Other Strategies

Concerning comparison, we have identified the following existing strategies that support pairwise testing: AETG (Cohen et al., 1997, Cohen et al., 1994), AETGm (Cohen, 2004), IPO (Lei & Tai, 1998), SA (Yan & Zhang, 2006), GA (Shiba et al., 2004), ACA (Shiba et al., 2004), and All Pairs tool (Bach, 2009). We considered 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 11 shows the size of the test set generated by each strategy, and Table 12 shows the execution time for each system. All the problem instances and data for the existing strategies are taken from (Younis et al., 2008), except for the All Pairs tool (which is free for download, hence, we could 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 summarized 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 11, G2Way and All Pairs generate the same number of test cases for S1. For S2, AETG, IPO, SA, GA, and ACA outperform G2Way and All Pairs. For S3, AETG gives the best result compared to all other strategies. For S4, G2Way comes second to ACA. For S5, G2Way outperforms IPO and All Pairs (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 results.

From the above given results, it can be seen that no strategy 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 results with small configurations, but appears to generate more test set with high configurations. Perhaps, All Pairs can be comparable to G2Way as it gives similar number of test sets for small configurations. However, G2Way appears to give better results for high configurations 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 managed 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 to other strategies (see Table 12).

Table 11

Comparison Based on the Number of 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 12

Comparison Based on 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

As discussed earlier, the fact that G2Way supports both the test data generation and execution can significantly influence its execution time as compared to other strategies. In G2Way, the base test data need to be specified in the external file, that is, to permit the specification of real data values for consideration (as opposed to merely symbolic values in order to permit execution). In fact, unlike other strategies (e.g. All Pairs, IPO), G2Way permits the use of real data values as opposed to symbolic representation.

Notwithstanding the aforementioned variation in terms of input handling as well as the differences in the computing environment, it is clear that IPO outperformed other strategies as far as execution time is concerned. This may be due to the fact that it is impractical to hard code the base test data for generation. In this manner, there is bound to be some timing overhead in that IPO is deterministic algorithm and needs only one run. For this 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.

CONCLUSION

In this paper, we have proposed and implemented a deterministic computational strategy for pairwise testing, called G2Way, as well as demonstrated its correctness. Compared to other strategies, our evaluation results are encouraging with acceptable test size and execution time. In fact, G2Way is the only strategy that can support both test planning and automated (concurrent) test execution. In this manner, G2Way can potentially alleviate the mundane tasks as far execution of the combinatorial test data is concerned.

Concerning our evaluation of pairwise testing as a whole, we believe that much effort needs to be invested to develop strategies that can support higher order interaction. In fact, there is also a need for a systematic test-characterization exercise in a case-by-case basis before one can establish whether or not pairwise or higher order interactions can be effectively used for testing a particular programme. This finding is supported by our case studies involving the FileChooserDemo and the *college_acceptance* programme. Here, using pairwise generated test data, the FileChooserDemo achieves a good coverage (see Table 7) whilst the *college_acceptance* programme gives poor coverage (see Table 10) implying contradicting parameter interactions.

Acknowledgement

This research is funded by the USM Short Term grant – Development of Interaction Testing Tool for Pairwise Coverage with Seeding and Constraint.

REFERENCES

EMMA: a free Java code coverage tool. (2006) Retrieve from http://emma.sourceforge.net/

Bach, J. (2009). All Pairs test case generation tool. Retrieve from http://tejasconsulting.com/open-testware/feature/allpairs.html.

Cohen, D. M., Dalal, S. R., Fredman, M. L., & Patton, G. C. (1997). The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering, 23***,** 437–444.

Cohen, D. M., Dalal, S. R., Kajla, A., & Patton, G. C. (1994). The automatic efficient test generator *(AETG)* system. *Proceedings of the 5th International Symposium on Software Reliability Engineering.* Monterey, CA, USA.

Cohen, M. B., Gibbons, P. B., Mugridge, W. B., & Colbourn, C. J. (2003). Constructing test suites for interaction testing. *Proceedings of the 25th International Conference on Software Engineering.* Portland, Oregon USA.

Colbourn, C. J., Cohen, M. B., & Turban, R. C. (2004). A deterministic density algorithm for pairwise interaction coverage. *Proceedings of the IASTED International Conference on Software Engineering.* Innsbruck, Austria.

Copeland, L. (2004). *A Practitioner's guide to software test design.* Boston, MA, Artech House.

Grindal, M., Offutt, J., & Andler, S. F. (2005). Combination testing strategies: A survey. *Software Testing Verification and Reliability, 15***,** 167–200.

Hartman, A., & Raskin, L. (July, 2004). Problems and algorithms for covering arrays. *Discrete Mathematics-Elsevier, 284***,** 149–156.

Hedayat, A. S., Sloane, N. J. A., & Stufken, J. (1999). *Orthogonal arrays: Theory and applications.* New York: Springer Verlag.

Klaib, M. F. J., Zamli, K. Z., Isa, N. A. M., Younis, M. I., & Abdullah, R. (2008). G2Way – A backtracking strategy for pairwise test data generation. *Proceedings of the 15th IEEE Asia-Pacific Software Engineering Conference.* Beijing, China.

Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., & Lawrence, J. (2007). IPOG: A general strategy for T-Way software testing. *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems.* Tucson, AZ, USA.

Lei, Y., & Tai, K. C. (1998). In-Parameter-Order: A test generation strategy for pairwise testing. *Proceedings of the 3rd IEEE International High-Assurance Systems Engineering Symposium* Washington, DC, USA.

Schroeder, P. J., & Korel, B. (2000). Black-box test reduction using input-output analysis. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* Portland, OR, USA.

Shiba, T., Tsuchiya, T., & Kikuno, T. (2004). Using artificial life techniques to generate test cases for combinatorial testing. *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC).* Hong Kong, IEEE Computer Society.

SUN How to use file choosers. Retrieve from http://java.sun.com/docs/books/tutorial/uiswing/ components/filechooser.html

Tai, K. C., & Lei, Y. (2002). A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering, 28***,** 109–111.

Williams, A. W., & Probert, R. L. (1996). A practical strategy for testing pair-wise coverage of network interfaces. *Proceedings of the 7th International Symposium on Software Reliability Engineering.*

Yan, J., & Zhang, J. (2006). Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC).* Chicago.

Younis, M. I., Zamli, K. Z., & Isa, N. A. M. (2008). IRPS–An efficient test data generation strategy for pairwise testing. *Proceedings of the 12th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES).* Zagreb, Croatia, Springer-Verlag.

Zamli, K. Z., Isa, N. A. M., Klaib, M. F. J., & Azizan, S. N. (2007). A tool for automated test data generation (and execution) based on combinatorial approach. *International Journal of Software Engineering and Its Applications, 1*, 19–34.