Assessing IRPS as an efficient pairwise test data generation strategy

Mohammed I. Younis, Kamal Z. Zamli*,
Mohammad F.J. Klaib,
Zainal Hisham Che Soh,
Syahrul Afzal Che Abdullah
and Nor Ashidi Mat Isa

School of Electrical and Electronic Engineering,
Software Engineering Research Group,
Universiti Sains Malaysia,
14300 Nibong Tebal, Penang, Malaysia
E-mail: younismi@gmail.com
E-mail: eekamal@eng.usm.my
E-mail: mom_klaib@yahoo.com
E-mail: myzainalhisham@yahoo.com.sg
E-mail: bekabox@gmail.com
E-mail: ashidi@eng.usm.my
*Corresponding author

Abstract: This paper discusses a novel pairwise test data generation strategy, called Intersection Residual Pair Set Strategy (IRPS), based on an efficient data structure implementation. In doing so, this paper also demonstrates the correctness of IRPS as well as compares its effectiveness against the existing strategies including Automatic Efficient Test Generator (AETG) and its variations, In Parameter Order (IPO), Simulated Annealing (SA), Genetic Algorithm (GA), Ant Colony Algorithm (ACA), All Pairs, G2Way and Jenny. Empirical results demonstrate that IRPS, in most cases, outperforms other strategies as far as the number of generated test data and the execution time are concerned.

Keywords: pairwise testing; test planning; software testing; test automation.

Reference to this paper should be made as follows: Younis, M.I., Zamli, K.Z., Klaib, M.F.J., Soh, Z.H.C., Abdullah, S.A.C. and Isa, N.A.M. (2010) 'Assessing IRPS as an efficient pairwise test data generation strategy', *Int. J. Advanced Intelligence Paradigms*, Vol. 2, No. 1, pp.90–104.

Biographical notes: Mohammed I. Younis received a BSc Degree in Computer Engineering from Baghdad University, 1997, and MSc Degree from the same university in 2001. Currently, a PhD candidate in the area of software testing automation. His research interests include software engineering, parallel processing, embedded system and RFID development.

Kamal Z. Zamli obtained a BSc Degree in Electrical Engineering from WPI, USA, 1992, and MSc Degree (Real Time Software Engineering) from Universiti Teknologi Malaysia, 2001, and PhD degree in Software Engineering from University of Newcastle upon Tyne, UK, 2003. He is currently a Senior Lecturer and lecturing at the School of Electrical and Electronics Engineering, USM Engineering Campus in Nibong Tebal, Penang, Malaysia. His research interests include software engineering, software testing automation, and algorithm design.

Mohammad F.J. Klaib received a BSc Degree in Electronic Engineering from Al-Quds University in Jurusalem in 2002, and MSc in Computer Engineering from Near East University, Cyprus, 2004. Currently, a PhD candidate in the area of testing automation and combinatorial testing.

Zainal Hisham Che Soh received a BSc Degree in Electronic Engineering from University of Leeds, UK in 1997 and MSc from Universiti Teknologi Malaysia (UTM), Malaysia in 2003.

Syahrul Afzal Che Abdullah received BSc Degree in Electronic Engineering from University of Southampton, UK in 1997 and MSc from Universiti Teknologi Malaysia (UTM), Malaysia in 2003.

Nor Ashidi Mat Isa obtained his BEng in Electrical Engineering from Universiti Sains Malaysia in 2000 and PhD in Image Processing and Neural Networks from the same university in 2003. He specialises in the area of intelligent system, image processing, neural networks for medical applications and algorithms.

1 Introduction

Our continuous dependencies on software to assist as well as facilitate our daily chores often raise dependability issue particularly when software is being employed in 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. Given limited time and resources, it is often impossible to exhaustively consider all of these combinations. Thus, a sampling strategy is needed to select a subset of these combinations in a systematic manner.

Combinatorial explosion problem (Cohen et al., 1996; Zamli et al., 2007) poses one of the biggest challenges in modern computer science because 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 owing to the need for faster and higher capability processors along state-of-the-art computer hardware. Apart from parallel testing, systematic random testing (Yan and Zhang, 2006) 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. Earlier work suggests that pairwise sampling strategy (i.e., based on two-way parameter interaction) can be effective to uncover between 60% and 80% of faults (Kuhn et al., 2004; Lei et al., 2007). Here, any two combinations of parameter values are to be covered by at least one test (Cohen et al., 1996; Lei and Tai, 2002). 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 parameters and quadratically with the number of values (Cohen et al., 1996; Cohen, 2004). Motivated by such a challenge, we have developed an efficient pairwise test data generation strategy, called IRPS. IRPS is our research vehicle to investigate efficient strategy and data structure implementation to generate optimal pairwise test set that can eventually be generalised for higher order interactions as far as software testing is concerned.

This paper is organised as follows. Section 2 highlights the related work. Section 3 describes the IRPS in detail. Section 4 highlights our evaluation including the proof of correctness, a case study to show its effectiveness in testing, as well as comparison against existing strategies in terms of the execution time as well as the number of generated test data, and gives further optimisation to IRPS (Younis et al., 2008b). Finally, Section 5 gives our conclusion and suggestion for future work.

2 Related work

Existing strategies can be categorised into two dominant approaches, i.e., algebraic approaches and computational approaches (Lei et al., 2007).

Algebraic approaches construct test sets using predefined rules. Most algebraic approaches compute test sets directly by a mathematical function (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, algebraic approaches often impose restrictions on the system configurations to which they can be applied (Lei et al., 2007; Yan and Zhang, 2006). In a nut shell, algebraic approaches are often based on the extensions of the mathematical methods for constructing Orthogonal Arrays (OAs) (Bush, 1952; Mandl, 1985) and Covering Arrays (CAs) (Hartman and Raskin, 2004; Zekaoui, 2006). Some variations of the algebraic approach also exploit recursion to permit the construction of larger test sets from smaller ones (see Williams and Probert, 1996; Maity and Nayak, 2005).

Unlike algebraic approaches, computational approaches often rely on the generation of the all pair combinations. On the basis of all pair combinations, the computational approaches iteratively search the combinations space to generate the required test case until all pairs have been covered. Unlike algebraic approaches, the computational approaches can be applied to arbitrary system configurations. Nevertheless, in the case where the number of pairs to be considered is significantly large, adopting computational approaches can be expensive owing to the need to consider explicit enumeration from all the combination space.

Adopting the computational approaches as the main basis, an AETG (Cohen et al., 1997) and its variant (AETG2) employ a greedy algorithm to construct the test case, i.e., 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., 2004). Other variants to AETG that use stochastic greedy algorithms are GA and ACA (Shiba et al., 2004). In some cases, they give optimal solution than original AETG, although they share the common characteristic as far as being non-deterministic in nature.

IPO strategy (Lei and Tai, 1998, 2002) 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. All Pairs and Jenny strategies (i.e., downloadable tools) appear to share the same property as far as producing deterministic test cases is concerned although little is known about the actual strategies employed owing to limited availability http://www.satisfice.com, http://www.Burtle burtle.net/bob/math and Copeland (2004).

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 insignificant effect on the system (i.e., do not care), then the strategy randomly selects the appropriate replacement of the do-not-care value 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.

As far as other non-greedy strategies are concerned, some approaches opted to adopt heuristic search techniques such as hill climbing and SA (Yan and Zhang, 2006). Briefly, hill climbing and SA strategies start from some known test set. Then, a series of transformations were applied (starting from the known test set) until an optimum set is reached to cover all the pairwise combinations (Yan and Zhang, 2006). Unlike AETG and IPO, which build a test set from scratch, heuristic search techniques can predict the known test set in advance. As such, heuristic search techniques can produce smaller test sets than AETG and IPO, but they typically take longer time to complete (Lei et al., 2007).

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 exploits row indexes to facilitate the storing and searching of pairs, the technique similar to IPOG. The backtracking algorithm iteratively traverses the pairwise sets to combine pairs with common parameter values to complete a test suite (Klaib et al., 2008).

3 IRPS Background

Adopting the computational approaches as its basis, the IRPS for generating pairwise test data set takes the following steps:

Step 1: Generates all pairs and stores them into compact linked list called *Pi*.

Step 2: Searches the *Pi* list and takes the desired weight of the candidate case as a test case then deletes it from the *Pi* list.

Step 3: Repeats step 2 until the *Pi* list is empty.

As indicated earlier, the generated pairs are stored in compact linked list called *Pi*, which is a linked list of linked lists. For a test set with N parameters, the *Pi* list contains $(N-1)$ linked list. Each linked list contains nodes equal to the number of values defined by its parameter as well as an array of linked list that represents the pair of all other variables in the next linked lists.

To understand how the *Pi* list works, consider a four-3-valued parameters system (see Table 1): A = {a0, a1, a2}, B = {b0, b1, b2}, C = {c0, c1, c2} and D {d0, d1, d2}. In this example, we have $\binom{4}{2}3^2 = 54$ possible pairs of combinations.

Table 1 Example for four parameters with 3-valued inputs

A	*B*	*C*	*D*
a0	b0	c0	d0
a1	b1	c1	d1
a2	b2	c2	d2

In this case, the complete *Pi* linked list can be visualised as in Table 2 given earlier. Node a0 with the pairs linked list array contains the following pairs (<a0, b0>, <a0, b1>, <a0, b2>, …, <a0, d2>). Here, this list contains only pairs that are based on a0. Similarly, the same observation can be seen with other nodes in the lists. The significance of such arrangement is the fact that less storage unit is required when compared with storing all pairs in clear pairwise combinations. Considering the aforementioned example and assuming each variable takes a unit of storage, then arranging in clear pairwise combinations would require $(54 \times 2 = 108)$ storage unit. Using similar calculation, adopting our arrangement strategy requires merely $3 + (3 \times 9) + 3 + (3 \times 6) + 3 + (3 \times 3) = 63$ storage unit.

Table 2 *Pi* Linked list for storing combination pairs for four 3-valued parameters

(*index*) *i = 0*	*i = 1*	*i = 2*
a0	b0	c0
b0blb2	c0c1c2	d0d1d2
c0clc2	d0d1d2	
d0dld2		
a1	b1	c1
b0b1b2	c0c1c2	d0d1d2
c0c1c2	d0d1d2	
d0d1d2		
a2	b2	c2
b0b1b2	c0c1c2	d0d1d2
c0c1c2	d0d1d2	
d0d1d2		

To describe the IRPS in detail, it is necessary to define a number of terminologies. The *weight* of the candidate test case is defined as the number of pairs that are covered by that candidate. For example, the test case combination of a0b0c0d0 covers the pairs (<a0, b0>,<a0, c0>,<a0, d0>,<b0, c0>,<b0, d0>, and <c0, d0>) and the variables b0,c0,d0 in node a0, c0,d0 in node b0, and finally d0 in node c0, so its weight = 6. *The maximum weight*, wmax, for N parameters can be calculated by the following:

$$\text{wmax} = N \times (N - 1)/2.$$

Here, if $N = 4$, then wmax = $4 \times 3/2 = 6$. *The miss variable* is defined as the difference between the maximum weight and the weight of the candidate test case. *The intersection of node* in the list i with the list $(i + 1)$ is defined as the intersection between the node and all nodes given by the first row. IRPS constructs a double linked list that stores the original i node and the intersection with the second node in $i + 1$ list, as well as the rest of the nodes. If the first row in the pairs array is empty, the intersection process will be performed with all values of the nodes in the next list and the miss variable is reduced by one (if miss > 0). Otherwise, the intersection process will be terminated and the iteration moves to the next node. *The candidate test* case is obtained by taking the node value in each node in the double linked list. For the last node, the candidate test case takes the current value and the first element in the pair array. The candidate test case is taken as a test case only if its weight satisfies the desired weight criteria. If not, the intersection process will continue with the other nodes in the list (by deleting the last node in the double linked list and replace it with the intersection with next node in the list, or when there is no next node in the list, the strategy will delete the last two nodes and continue with the iteration). In other words, the intersection process goes horizontally when the target weight is not found and grows vertically in recursive fashion. Finally, the *delete operation* operates by deleting each variable (if they exist) in each node.

Figure 1 depicts the search algorithm for the proposed IRPS. Here, the algorithm is terminated whenever the *Pi* list is empty to guarantee that all pairs are covered and each pair only appears at most once in the final generated test cases (i.e., to achieve optimum solution).

Figure 1 The search algorithm

```
for (i=0;i<N-;i++) // i is the index of pi list
begin    //start the search with maximum weight
  w=N(N-1)/2;
  while (list(i) is not empty
    begin
      if (there exist candidate test case  from the intersection of  a node in ith
          List with the remaining i+1 ,...,N-1 Lists)
          delete the test case from pi list;
      else //not find a test case with the desired weight so :
          w--; //decrease the weight
  end
end
```

Referring to our earlier example with four parameters and three values, the test set is generated with its weight and miss values using IRPS are given in Table 3.

Table 3 The generated test set

No.	*Test case*	*Miss*	*Weight*
T1	a0b0c0d0	0	6
T2	a0blcldl	0	6
T3	a0b2c2d2	0	6
T4	a1b0c1d2	0	6
T5	a1b1c2d0	0	6
T6	a1b2c0d1	0	6
T7	a2b0c2d1	0	6
T8	a2b1c0d2	0	6
T9	a2b2c1d0	0	6

4 Evaluation

Our evaluation has four main goals. First, we want to demonstrate the correctness of IRPS. Second, we want to demonstrate the use of IRPS for system-level testing. Third, we intent to investigate the growth in the size of the test sets generated by IRPS, as well as the time taken to produce those test sets based on the given number of parameters and values. In doing so, we suggest adding artificial parameters and values method to ensure optimal test case. Finally, we want to compare the performance of IRPS against existing tools particularly in terms of the size and the time taken to produce the test sets. In the next subsections, we will present our complete evaluations based on the aforementioned goals.

4.1 Demonstration of correctness

In this section, we will focus on demonstrating the correctness of the IRPS by analysing the resulting test case set. Here, we aim to show that IRPS gives optimum results, i.e., all pairs of combinations are covered at least once. To achieve this goal, we developed IRPS_Prover algorithm that works as given in Figure 2.

Figure 2 IRPS_Prover algorithm

Algorithm *IRPS-Prover (Test Case ts, ParameterSet ps)*
{
1. denote the parameters in ps, in an arbitrary order, as P1, P2, …, and Pn
2. ***for*** *(int i =1; i < n; i ++){*
3. ***for*** *(int j=i+1; j ≤ n; j ++){*
4. let π be the set of Pairwise combinations of values involving parameter Pi and Pj parameter, with empty location and zero number of occurrence. }//loop j
5. }//loop i
6. ***for*** *(each test τ in test set ts) {*
7. Decompose τ into list of Pairs, then locates each pair in π, indicates τ, and increments the number of occurrence}
8. Display the occurrence table
}

Table 4 shows the pairs of combination, corresponding location in the generated test case (case) and finally the number of appearance (#) for the generated test case given in Table 3.

From the table, we observe that each combination pair appears exactly one (which means that the number of generated test cases is optimal number) and there is no missing pair (which means that our strategy is correct).

Table 4 The location of each pair in the generated test case and its appearance

Pairs	*Case*	#	*Pairs*	*Case*	#	*Pairs*	*Case*	#
a0, b0	T1	1	a1, b0	T4	1	a2, b0	T7	1
a0, c0	T1	1	a1, c0	T6	1	a2, c0	T8	1
a0, d0	T1	1	a1, d0	T5	1	a2, d0	T9	1
a0, b1	T2	1	a1, b1	T5	1	a2, b1	T8	1
a0, c1	T2	1	a1, c1	T4	1	a2, c1	T9	1
a0, d1	T2	1	a1, d1	T6	1	a2, d1	T7	1
a0, b2	T3	1	a1, b2	T6	1	a2, b2	T9	1
a0, c2	T3	1	a1, c2	T5	1	a2, c2	T7	1
a0, d2	T3	1	a1, d2	T4	1	a2, d2	T8	1
b0, c0	T1	1	b1, c0	T8	1	b2, c0	T6	1
b0, d0	T1	1	b1, d0	T5	1	b2, d0	T9	1
b0, c1	T4	1	b1, c1	T2	1	b2, c1	T9	1
b0, d1	T7	1	b1, d1	T2	1	b2, d1	T6	1
b0, c2	T7	1	b1, c2	T5	1	b2, c2	T3	1
b0, d2	T4	1	b1, d2	T8	1	b2, d2	T3	1
c0, d0	T1	1	c1, d0	T9	1	c2, d0	T5	1
c0, d1	T6	1	c1, d1	T2	1	c2, d1	T7	1
c0, d2	T8	1	c1, d2	T4	1	c2, d2	T3	1

4.2 Effectiveness of IRPS

In accessing the effectiveness of the IRPS, it is important to have a well-developed third-party software having access to their internal source code. An independent open-source code named as the FileChooserDemo programme (SUN Microsystems, http://java.sun.com/docs /books/tutorial/uiswing/ components/filechooser.html) has been chosen to demonstrate the effectiveness of the IRPS for pairwise test data generation (i.e., downloadable from the SUN Microsystem website). The FileChooserDemo is a programme to show various Java GUI for selection-based controls (see Figure 3).

The FileChooserDemo programme has 14 input parameters (one 4-valued parameters, two 3-valued parameters, 11 2-valued parameters) as shown in Table 5 and the detail of each input parameter value is illustrated in Table 6.

Figure 3 FileChooserDemo interface (see online version for colours)

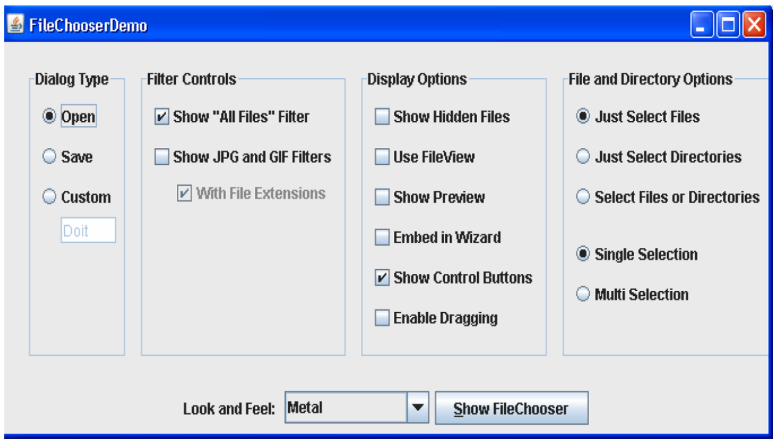

Table 5 The input parameters and the number of each input parameters value for the FileChooserDemo

Parameter	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	*P13*	*P14*
Number of values	4	3	3	2	2	2	2	2	2	2	2	2	2	2

Table 6 The detail input parameters and the detail of each input parameters value for the FileChooserDemo

Input parameter	*Parameter value 1*	*Parameter value 2*	*Parameter value 3*	*Parameter value 4*
P1 – Look and feel	Metal	Open	Just select files	Checked
P2 – Dialogue type	CDE/motif	Save	Just select directories	Not checked
P3 – File and directory options	Windows Checked	Custom not checked	Just select files or directories	–
P4 – Show ‘all files’ filter	Checked	Not checked	–	–
P5 – Show JPG and GIF filters	Checked	Not checked	–	–
P6 – With file extensions	Checked	Not checked	–	–
P7 – Show hidden files	Checked	Not checked	–	–
P8 – Use file view	Checked	Not checked	–	–
P9 – Use preview	Checked	Not checked	–	–
P10 – Embed in wizard	Checked	Not checked	–	–
P11 – Show control buttons	Checked	Not checked	–	–
P12 – Enable dragging	Checked	Not checked	–	–
PI3 – File and directory options	Single Selection	Multi selection	–	–
P14 – Show file chooser	Select	Cancel	–	–

On the basis of the number of parameters, considering an exhaustive testing would require $4^1 \times 3^2 \times 2^{11} = 73728$ test cases to be tested. Now, using pairwise testing and applying the IRPS, the test cases are reduced to merely 18 as shown in Table 7.

Table 7 Suggested test suite

No.	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*	*P11*	*P12*	*P13*	*P14*
1	M	O	J.S.F	T	T	T	T	T	T	T	T	T	SS	S
2	M	S	J.S.D	F	F	F	F	F	F	F	F	F	MS	C
3	CDE/M	C	F or D	T	F	F	T	T	F	F	T	T	MS	C
4	W	C	F or D	F	T	T	F	F	T	T	F	F	SS	S
5	WC	O	J.S.F	F	F	F	F	T	T	T	T	F	MS	C
6	WC	S	J.S.D	T	T	T	T	F	F	F	F	T	SS	S
7	CDE/M	O	J.S.F	F	T	T	F	F	F	F	F	T	MS	C
8	W	S	J.S.D	T	F	F	T	T	T	T	T	F	SS	S
9	CDE/M	S	J.S.D	T	T	F	F	T	T	T	F	F	SS	S
10	W	O	J.S.F	F	F	T	T	F	F	F	T	T	MS	C
11	M	C	F or D	F	T	F	T	F	T	F	T	F	SS	C
12	WC	C	F or D	T	F	T	F	T	F	T	T	F	MS	S
13	CDE/M	O	J.S.D	F	T	F	T	F	F	F	T	T	MS	C
14	WC	O	F or D	T	T	F	T	T	T	F	F	T	SS	S
15	M	S	J.S.F	T	F	T	F	F	F	T	F	F	SS	S
16	W	S	F or D	F	F	F	F	T	F	T	T	T	MS	S
17	CDE/M	C	J.S.F	F	T	T	T	F	T	F	F	T	MS	C
18	M	C	J.S.D	T	F	T	F	T	T	T	F	F	SS	C

From the generated test case, we are going to investigate whether the 18 suggested test cases are sufficient to test FileChooserDemo programme whilst giving acceptable testing 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 the programme behaves as expected.

To measure coverage, we have utilised an open-source test coverage tool known as EMMA (2006), from Source Forge. Using EMMA, a number of coverage metrics can be reported such as class coverage, method coverage, block coverage and line coverage. First, the class coverage refers to the ratio of the covered classes over the total number of classes. As for the second coverage metric, 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. Lastly, the line coverage is defined as the covered lines over the total number of lines. During the execution of the 18 suggested test cases, we do not detect any error and the programme output is as expected. Using EMMA, we obtain the following coverage results as shown in Table 8. These metrics are calculated based on the FileChooserDemo implementation consisting of 9 classes, 42 methods, 2136 blocks and 450 lines.

Referring to the coverage results tabulated in Table 8, it can be deduced that the pairwise test data set generated by IRPS is reasonably effective in covering 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 cannot be detected even with exhaustive combinations). Thus, we conclude that IRPS is an effective pairwise testing strategy for detecting and covering all the coverage metric for FileChooserDemo program.

Table 8 Percentage coverage

Class coverage (%)	*Method coverage* (%)	*Block coverage* (%)	*Line coverage* (%)
100	83	96	94

4.3 IRPS behaviour and its enhancement in terms of test size and execution time

To perform the evaluation, we have applied IRPS to three series of system configurations. In the first series, the number of parameters (p) and the number of variables (v) are equal to each other, the numbers (n) are (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 and 16), respectively. In the second series, the number of parameters is fixed to be 5, and the number of variables is varied from 2 to 10.

Tables 9–11 show the experimental results for the three series of system configurations, respectively. The columns in the three tables are self-explanatory. Note that the execution times are shown in seconds, and all the results were collected using a laptop running Windows Vista with 1.6 GHZ CPU and 512 MB memory. The entire tool is implemented using Java Development Kit 1.4 (JDK1.4) platforms.

For pairwise interaction, the optimal size can be viewed as the product of the two maximum numbers of variables. This observation can be seen in the case of CA1, CA2, CA3, CA4, CA6, CA7 and CA10 from Table 9. Similar observation can be seen in the case of CA13, CA14 and CA16 from Table 10. The generated test case is also minimal in size, as depicted in CA20, CA21, CA22, CA23 and CA24 from Table 11, respectively. Here, we conclude that the size of generated test case depends linearly on the optimal size of the generated test case.

Table 9 Results for $n = 2$ to 11 n n-valued parameters

Case name	*CA1*	*CA2*	*CA3*	*CA4*	*CA5*	*CA6*	*CA7*	*CAS*	*CA9*	*CA10*
$N = p = v$	2	3	4	5	6	7	8	9	10	11
Size	4	9	16	25	44	49	64	116	149	121
Time	<0.001	<0.001	0.011	0.015	0.087	0.034	0.077	240.2	16.35	0.121

Table 10 Results for five parameters with 2–10 values

Case name	*CA11*	*CA12*	*CA13*	*CA14*	*CA15*	*CA16*	*CA17*	*CA18*	*CA19*
Value (v)	2	3	4	5	6	7	8	9	10
Size	6	12	16	25	44	49	78	96	114
Time	0.01	0.015	0.016	0.015	0.077	0.057	0.133	0.178	0.184

Table 11 Results for 2–10 parameters with 5 values

Case Name CA20	*CA21*	*CA22*	*CA23*	*CA24*	*CA25*	*CA26*	*CA27*	*CA28*
Parameter (*p*) 2	3	4	5	6	7	8	9	10
Size 25	25	25	25	25	37	41	44	45
Time 0.053	0.054	0.114	0.015	0.031	0.32	0.78	1.45	1.928

As far as execution time is concerned, we observe that the execution time is significantly independent on the number of parameters and values when the size is not minimal. This is due to the nature of the algorithm that generates the heavy-weighted test case first, deletes them from the *Pi* list, and then searches again for the uncovered pairs. In this way, the size of the generated test case and the execution time depend on the phenomena of greedy algorithm rather than the number of parameters and values.

We observe that the size and execution time of CA9 (10 10-valued parameters) is greater than CA10 (11 11-valued parameters), according to Table 9, and the size of CA17 (five 8-valued parameters) is greater than CA7 (eight 8-valued parameters) according to Tables 9 and 10, respectively. Here, we conclude that the behaviour of IRPS is unpredictable in term of the execution time owing to the exhaustive search nature when drifting from optimal size, but running the test case generator produces the same test set on every case (thus, IRPS is deterministic). This gives us the motivation for further optimisation by introducing adding artificial variables and parameters in generation IRPS test case catalogue. The complete discussion with illustrative example is given in Younis et al. (2008a). Thus, by applying IRPS_RA and IRPS_ORA (Younis et al., 2008a), the test case CA9 is built fom CA10 and the size will be 120 and 118, respectively. Similarly, we built CA17 from CA7 and the size will be 64 test cases only.

4.4 Comparison with other strategies

As for comparison, we have identified the following existing strategies that support pairwise testing: AETG (Cohen et al., 1996, 1997), AETG2 (Shiba et al., 2004; Cohen et al., 2003), IPO (Lei and Tai, 2002), SA (Shiba et al., 2004), GA (Shiba et al., 2004), ACA (Shiba et al., 2004) and All Pairs tool (http://www.satisfice.com). We consider eight systems namely S1: three 3-valued parameters, S2: four 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 and S8: one 5-valued parameters, eight 3-valued parameters and two 2-valued parameters. The system configurations are AETG2 & SA: C++, Linux, Intel P IV 1.8 GHZ; IPO: Java, Windows 98, Intel P II 450 MHZ; CA, & ACA: C, Windows XP, P IV 2.26 GHZ; All Pairs: Perl, Windows Vista, P IV 1.6 GHZ, 512 MB RAM; G2Way, Jenny: Intel P IV 1.8 Ghz, 512 MB RAM, C++ programming language, Windows Vista operating system; IRPS: Java, Windows Vista, P IV 1.6 GHZ, 512 MB RAM.

Table 12 shows the size of the test set generated by each strategy, and Table 13 shows the execution time for each system. All the problem instances and data for the existing strategies are taken from Lei and Tai (2002), Shiba et al. (2004) and Cohen et al. (2003) except for All Pairs tool (available freely, which we run side by side with our tool). Entries marked with NA are data that are not available in these papers.

Referring to Table 12, IRPS always generates smaller test cases than ALL Pairs and in some cases generates less (i.e., S4, S5, S6 and S7) or equals to that of IPO (i.e., S2, S3). IRPS also generates less cases compared with AETG2 (except S6), GA and ACA (except S8). Whereas IRPS outperformed AETG in S8, AETG outperformed IRPS in S3 and S6. Finally, SA outperformed IRPS (in S3, S6 and S8). Unlike AETG, AETG2, GA, ACA and IRPS, SA does not have the practical advantage of the greedy algorithm, as the implementation is not based on such an algorithm. Here, in the absence of the greedy algorithm, the construction of the test set cannot utilise the useful property that the test case created earlier has more significant impact as far as the interaction coverage is concerned (Shiba et al., 2004).

Table 12 Comparison on the size of the test set generated by existing strategies

System	*AETG*	*AETG2*	*IPO*	*SA*	*GA*	*ACA*	*ALL Pairs*	*G2Way*	*Jenny*	*IRPS*
S1	NA	NA	NA	NA	NA	NA	10	10	9	9
S2	9	11	9	9	9	9	10	10	13	9
S3	15	17	17	16	17	17	22	19	20	17
S4	NA	NA	169	NA	157	159	177	160	157	149
S5	NA	NA	361	NA	NA	NA	390	343	336	321
S6	180	198	212	183	227	225	230	200	194	210
S7	NA	NA	47	NA	NA	NA	49	46	45	45
S8	19	20	NA	15	15	16	21	23	23	17

Admittedly, no fair comparison can be made in terms of execution time from existing strategies owing to the differences in the computing environments, and the unavailability of the open-source code or executable code to run in our platform (with the exception of ALL Pairs tool). Nevertheless, as a general observation, we believe that the execution time for IRPS is still acceptable when compared with other strategies (see Table 13). Not considering the computing differences, IPO outperforms all other strategies. One reason may be that IPO employs deterministic algorithm and needs only one run. Thus, IPO requires much less time to execute than others. SA includes the time taken to find all sized test sets through binary search process, hence, requiring more run time than others. In short, no strategies can clearly be dominant in all.

Table 13 Comparison on the time taken to generate test set (in seconds) for existing strategies

System	*AETG*	*AETG2*	*IPO*	*SA*	*GA*	*ACA*	*ALL Pairs*	*G2Way*	*Jenny*	*IRPS*
SI	NA	NA	NA	NA	NA	NA	0.08	0.047	0.19	<0.001
S2	NA	NA	NA	NA	NA	NA	0.23	0.062	0.2	0.004
S3	NA	NA	NA	NA	NA	NA	0.45	0.25	0.312	39.23
S4	NA	NA	0.3	NA	866	1180	5.03	2.906	0.43	16.35
S5	NA	NA	0.72	NA	NA	NA	10.36	7.438	2.392	1124
S6	NA	6001	NA	10,833	6365	7083	23.3	1753	3.33	3213
S7	NA	NA	0.05	NA	NA	NA	1.02	0.687	0.27	1.928
S8	NA	58	NA	214	22	31	0.35	0.33	0.251	2.02

5 Conclusion and future work

In this paper, we propose a novel deterministic computational strategy for pairwise testing with efficient data structure for storing and searching pairs. Our initial evaluation results are encouraging particularly in terms of test suite size within acceptable execution time in terms for optimality as well as comparisons with other strategies. Also, the paper includes a case study that demonstrates the ease of use of IRPS in the generation of test set. As part of our future work, we are currently investigating a new parallel search algorithm for IRPS to be implemented under the GRID environment. Also, our aim is to generalise IRPS to support N-way testing. Finally, we will support further test case minimisation by using input and output relationship.

Acknowledgements

This research is partially funded by the USM GRID – The Development and Integration of Grid Services and Applications. The first author, Mohamad I. Younis, is the USM fellowship recipient.

References

Bush, K.A. (1952) 'Orthogonal arrays of index unity', *Annals of Mathematical Statistics*, Vol. 23, pp.426–434.

Cohen, D.M., Dalal, S.R., Fredman, M.L. and Patton, G.C. (1997) 'The AETG system: an approach to testing based on combinatorial design', *IEEE Trans. on Software Engineering*, Vol. 23, No. 7, pp.437–443.

Cohen, D.M., Dalal, S.R., Parelius, J. and Patton, G.C. (1996) 'The combinatorial design approach to automatic test generation', *IEEE Software*, Vol. 13, No. 5, pp.83–88.

Cohen, M.B. (2004) *Designing Test Suites for Software Interaction Testing*, PhD Thesis, University of Auckland, Auckland, New Zealand.

Cohen, M.B., Gibbons, P.B., Mugridge, W.B. and Colbourn, C.J. (2003) 'Constructing test suites for interaction testing', *Proc. 25th Intl. Conf. on Software Engineering (ICSE' 03)*, IEEE CS Press, May, Dallas, USA, pp.38–48.

Copeland, L. (2004) *A Practitioner's Guide to Software Test Design*, STQE Publishing, Massachusetts, USA.

EMMA (2006) *EMMA: A Free Java Code Coverage Tool*, Available from: http://emma.sourceforge.net/

Grindal, M., Offutt, J. and Andler, S.F. (2004) 'Combination testing strategies: a survey', *GMU*, Technical Report ISE-TR-04-05 July.

Hartman, A. and Raskin, L. (2004) 'Problems and algorithms for covering arrays', *Discrete Mathematics*, Vol. 284, Nos. 1–3, pp.149–156.

Klaib, M.F.J., Zamli, K.Z., Isa, N.A.M., Younis, M.I. and Abdullah, R. (2008) 'G2Way – a backtracking strategy for pairwise test data generation', *The 15th Asia-Pacific Software Engineering Conference (APSEC 08)*, December, Beijing, China, pp.463–470.

Kuhn, D.R., Wallace, D.R. and Gallo, A.M. (2004) 'Software fault interactions and implications for software testing', *IEEE Trans. on Software Engineering*, Vol. 30, No. 6, June, pp.418–421.

Lei, Y. and Tai, K.C. (1998) 'In-parameter-order: a test generating strategy for pairwise testing', *Proc. 3rd IEEE Intl. Symp. on High Assurance System Engineering*, November, Washington DC, USA, pp.254–261.

Lei, Y. and Tai, K.C. (2002) 'In-parameter-order: a test generating strategy for pairwise testing', *IEEE Transaction on Software Engineering*, Vol. 28, No. 1, pp.1–3.

Lei, Y., Kacker, R., Kuhn, D.R., Okun, V. and Lawrence, J. (2007) 'IPOG: a general strategy for *t*-way software testing', *14th Annual IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems*, March, IEEE CS Press, Tucson, AZ, pp.549–556.

Maity, S. and Nayak, A. (2005) 'An improved test generation strategy for pair-wise testing', *Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*, Washington DC, USA, pp.235–244.

Mandl, R. (1985) 'Orthogonal Latin squares: an application of experiment design to compiler testing', *Communications of the ACM*, Vol. 28, No. 10, pp.1054–1058.

Schroeder, P.J. and Korel, B. (2000) 'Black-box test reduction using input-output analysis', *Proc. International Symposium on Software Testing and Analysis (ISSTA 2000)*, August, Portland, OR, USA, pp.21–24.

Shiba, T., Tsuchiya, T. and Kikuno, T. (2004) 'Using artificial life techniques to generate test cases for combinatorial testing', *28th Annual Intl. Computer Software and Applications Conference (COMPSAC'04)*, September, Hong Kong, China, pp.72–77.

Williams, A.W. and Probert, R.L. (1996) 'A practical strategy for testing pair-wise coverage of network interfaces', *Proc. 7th Intl. Symp. on Software Reliability Engineering (ISSRE)*, White Plains, New York.

Yan, J. and Zhang, J. (2006) 'Backtracking algorithms and search heuristics to generate test suites for combinatorial testing', *Proc. 30th Annual Intl. Computer Software and Applications Conference (COMPSAC'06)*, IEEE CS Press, September, Chicago, USA, Vol. 1, pp.385–394.

Younis, M.I., Zamli, K.Z. and Isa, N.A.M. (2008a) 'Generating pairwise combinatorial test set using artificial parameters and values', *The 3rd International Symposium on Information Technology, (ITSim'08)*, IEEE Press, August, KLCC, Malaysia, Vol. 3, pp.1654–1661.

Younis, M.I., Zamli, K.Z. and Isa, N.A.M. (2008b) 'IRPS – an efficient test data generation strategy for pairwise testing', *12th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems, (KES)*, September, Zagreb, Croatia, pp.493–500.

Zamli, K.Z., Isa, N.A.M., Klaib, M.F.J. and Azizan, S.N. (2007) 'Designing a combinatorial java unit testing tool', *Proc. IASTED Intl. Conference on Advances in Computer Science and Technology (ACST 2007)*, April, Phuket, Thailand, pp.153–158.

Zekaoui, L. (2006) *Mixed Covering Arrays on Graphs and Tabu Search Algorithms*, MSc Thesis, Ottawa-Carleton Institute for Computer Science, University of Ottawa, September, Canada.

Websites

http://www.burtleburtle.net/bob/math

http://www.satisfice.com

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