Scientific Research and Essays Vol. 6(16), pp. 3418-3430, 19 August, 2011
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.427
ISSN 1992-2248 ©2011 Academic Journals

Full Length Research Paper

Development of interaction test suite generation strategy with input-output mapping supports

H. Y. Ong and Kamal Z. Zamli*

School of Electrical and Electronic Engineering, Engineering Campus, Universiti Sains Malaysia, 14300 Nibong Tebal, Pulau Pinang, Malaysia.

Accepted 18 July, 2011

Software testing relates to the process of finding errors/defects and/or ensuring that a particular software of interest meets its specification. One of the key activities within software testing is on the test case design. Over the years, many test case design strategies have been developed in the literature including that of boundary values, equivalence partitioning, decision tables, robustness consideration as well as cost and effect graphing. Although useful, these strategies do not sufficiently cater for bugs due to interaction. Addressing the aforementioned issue, many researches into interaction based strategies, called *t*-way strategies (where *t* represents interaction strength), have started to emerge in the literature. This paper presents the development of a new *t*-way strategy called AURA. AURA strategy serves as our research vehicle to investigate the usefulness of automated mapping based on input-output relationship as well as its flexible iteration control for constructing *t*-way test suite. Benchmarking results demonstrate that AURA strategy gives competitive results against most existing strategies.

Key words: Interaction testing, *t*-way testing, variable strength interaction testing, input-output based relationship interaction testing, automated mapping support.

INTRODUCTION

Software can fail in many unexpected ways (Kimoto et al., 2008; Maity and Nayak, 2005). Thus, in order to ensure quality and conformance to specifications, there is a need to exhaustively test them. Yet, exhaustive testing is practically impossible. Addressing this issue, many test case design strategies have been developed in the literature (for example boundary value analysis, equivalence partitioning, decision tables and random testing (Basili and Selby, 1987; Beer and Mohacsi, 2008; Kuhn et al., 2004; Reid, 1997) to help sample out test data into manageable ones. Although useful, these strategies do not sufficiently cater for faults due to interaction. For this reason, *t*-way strategies have started to emerge. Numerous efficient *t*-way testing strategies have been proposed in the past literatures (Ahmed et al., 2011; Baowen et al., 2003; Jangbok et al., 2007; Klaib et al., 2008; Lei et al., 2007; Sunderam et al., 2005; Yingxia, 2009; Younis and Zamli, 2010) to generate optimized test cases for SUT. Meanwhile, most of the reported strategies that evolved earlier, enumerate their test cases by covering all *t*-interactions of parameters involved. These are known as uniform strength interaction testing since *t* is a fixed integer value in their consideration. However, *t* is rarely uniform in real world. Not all interaction faults from typical SUT are solely constituted by these *t*-interactions. In fact, a particular subset of parameters can have a higher degree of interaction than others which indicating failures due to the interaction of that subset may have more significant impact to the overall system (Myra et al., 2003a). For example, consider a subset of components that control a safety-critical hardware interface. A stronger coverage is needed in that area (that is t = 3) but the rest of the components may be sufficiently tested with pair-wise testing (that is t = 2). Therefore, variable strength interaction testing strategy is then been proposed to support this concern (Myra et al., 2003a, 2003b; Zamli

*Corresponding author. E-mail: eekamal@eng.usm.my, yeh_1985@hotmail.com. Tel: 604-5996079.

and Younis, 2010; Ziyuan et al., 2008).

Variable strength interaction testing no doubt solves some of real considerations by allowing certain subsets to cover higher *t*-interactions, though; it is still insufficient to generate test cases based on actual interactions (Wang et al., 2007). Also, variable strength interaction test suite is tightly coupled by its interaction strength. As such, variable strength interaction can be regarded as the combinations of uniform strength interaction testing in the smaller scale. As some of *t*-interactions are not responsible to actual interactions but still have to be taken into account on test cases generation, this strategy might include these redundant test cases. To overcome this limitation, recent studies (Patrick and Bogdan, 2000; Patrick et al., 2002; Wang et al., 2007; Zabil et al., 2011) claimed that interaction testing should focus on those input combinations that affect a program output, rather than considering all possible input combinations. Consequently, a general solution has been introduced: input-output based relationship interaction testing (Patrick and Bogdan, 2000). This strategy captures the actual interactions for SUT based on input-output relationship and is also capable to revert back and support both uniform and variable strength test suite generation. For this reason, developments of efficient input-output based relationship interaction testing strategies are preferable. As far as implementation is concerned, most existing strategy implementations (Ahmed and Zamli, 2010; Lei et al., 2007; Younis and Zamli, 2010; Yu and Kuo-Chung, 1998) generate their outputs in terms of symbolic parameters for ease of data manipulation. This could be straightforward but not user friendly approach because test engineers have to manually map these symbolic values to actual data one by one before they could execute on them. As the test case number is predominantly large especially in highly configurable software systems, these could be another problematic issue in term of time and cost consumed as well as the accuracy of test cases (due to the potential of human errors on manually mapping process). Hence, there is a need for automated input-output mapping to seamlessly translate the symbolic outputs back into the actual data form. Apart from automated input-output mapping, existing strategy implementations are also lacking as far as flexibility of test suite generation is concerned. Here, the problem of *t*-way test suite generation can be seen as two sides as the same coin with optimal size and test generation time being the sides. On one side of the coin, when the optimality of test suite size is preferred than generation time, a strategy need to be adaptable to generate more optimized test suite. On the other side of same coin, a strategy needs to be flexible enough to generate fast test suite but in expense of optimality. In order to address these aforementioned issues, we have developed a non-deterministic input-output based relationship *t*-way testing strategy, AURA.

AURA strategy serves as our research vehicle to investigate the usefulness of automated mapping based on input-output relationship as well as its flexible iteration control for constructing *t*-way test suite. Benchmarking results demonstrate that AURA strategy gives competitive results against most existing strategies. The rest of the paper is structured as follows: Subsequently the mathematical background on interaction testing is given after which the study discusses the recent related work on input-output based relationship interaction testing; whereas, the development of AURA strategy is further illustrated and clarified. Then, we compare and discuss our results with other existing strategies. Lastly, we conclude our work.

MATHEMATICAL BACKGROUND

Mathematically, interaction testing can be abstracted to a covering array (CA). CA is a combinatorial object that been extensively used to generate interaction test cases in software systems when all factors (parameters) have equal number of levels (options or values). A covering array, CA (N; t, k, v), is an array with N rows and k columns that satisfies the criteria that each t-tuple occurs at least once within these rows (Cemal et al., 2006; Myra, 2003). When N is unknown or unspecified, the notation CA (t, k, v) can be used (that is t is interaction strength, k is the number of factors and v is the number of options associated with each factor). For covering array, the value of v is the same for all k. Meanwhile, mixed-level covering array is a generalization of covering array that allows for different alphabet sizes for different rows. The mixed-level covering array is denoted as MCA (N; t, k, (v_1, v_2, …, v_k)), an N x k array on v symbols (Bryce and Colbourn, 2007; Yan and Jian, 2006), where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1) Each column I ($1 \leq I \leq k$) contains only elements from a set S_i with $| S_i | = v_i$.
2) The rows of each N x t sub-array cover all t-tuples of values from the t columns at least once.

A shorthand notation can be used to describe MCA (also for CA, VCA and IOR) by combining the same v_i's and representing this number as a superscript (Yan and Jian, 2006). For instance, three v_i's each with two options is written as 2^3. In this manner, an MCA (N; t, k, (v_1, v_2, …, v_k)) can also be written as an MCA (N; t, (s_1^{p1}, s_2^{p2}, …, s_r^{pr})) where $k = \sum_{i=1}^{r} p_i$. Variable strength covering array, denoted as VCA (N; t, (v_1, v_2, …, v_k), C), is an N x k mixed level covering array, of strength t containing C, a vector of covering arrays each of strength greater than t and defined on a subset of the k columns. Ordering of the columns in the representation of a VCA is important since the columns of the covering arrays in C are listed consecutively from left to right (Myra et al., 2003b; Ziyuan et al., 2008). Unlike CA, MCA and VCA, input-output based relationship covering array needs not generate test cases to cover all *t*-way interactions but only required to cover all actual interactions. This covering array can be denoted as IOR (*N*; (v_1, v_2, …, v_k), *R*), an *N* x *k* mixed level covering array which covers interaction relationship, *R*, of a typical software SUT. *R* is consisted of *w* number of interaction coverage requirement, *r*, which specified the actual interactions for that SUT and is defined as $R = \{r_1, r_2, ..., r_w\}$ (Patrick and Bogdan, 2000; Wang et al., 2007). Each *r* indicates a set of inputs (factors) that are interacting and is constitute to a specified interaction coverage requirement.

RELATED WORK

In the last 15 years, many *t*-way strategies have been proposed in the literature including automatic efficient test generator (AETG) (Cohen et al., 1997), pairwise independent combinatorial testing (PICT) (Czerwonka, 2006), in parameter order (IPO) and its variants (Lei et al., 2007; Reussner et al., 2005; Younis and Zamli, 2010; Yu and Kuo-Chung, 1998; Yu et al., 2008), genetic algorithm (GA) (McCaffrey, 2009a), simulated bee colony algorithm (SBC) (McCaffrey, 2009b), simulated annealing (SA) (Myra et al., 2003b) and ant colony system (ACS) (Xiang et al., 2009). All aforementioned strategies are found useful and become the pioneers in *t*-way and variable strength interaction testing. Moreover, a comprehensive survey of interaction testing has been published by Nie and Leung (2011) recently. However, in line of the scope of this paper, the further discussions shall be drawn on the recent works in input-output based relationship interaction testing. Considering the support of input-output based relationship interaction testing, much useful effort is also emerging. Patrick (2001) proposed the model of input-output based relationship testing method and gave three different test generation algorithms (Patrick, 2001; Patrick and Bogdan, 2000; Patrick et al., 2002) to solve the problem of test cases generation for software with complex input-output relationship. Their first approach implemented a brute force algorithm to explore all possible combinations of test to discover the correct minimal test suite (Patrick and Bogdan, 2000). Although straightforward, brute force algorithm tends to consume time especially involving large test data. Next, they proposed Union algorithm (Patrick, 2001) by generating a serial of test suite (that is, sets of test cases) for output variables to cover the interaction that is corresponding to their associative inputs variables, and then taking the union of them to obtain a final test suite.

The implementation of the Union algorithm is straightforward with low time complexity but generally not producing optimal test suite. Lastly, Patrick et al. (2002) proposed Greedy algorithm. The algorithm works by selecting an unused test case that covers the greatest number of uncovered combinations of input values each times until all interactions have been covered by the selected test suite. It indeed generate a much smaller test suite than Union algorithm, but with a bad time and space performance since this method has to check all test cases in a huge search space. To overcome this constraint, Cheng et al. (2003) implemented a problem reduction method which is based on color graph in their later work. This method only shows significant gain in efficiency (both time and space performance) when the number of colors used is small relative to the number of nodes in a complete graph. In other words, this reduction method is merely applicable for simple relationships between the input parameters based on their occurrences in the output parameters. Later on, Wang et al. (2007) analyzed and improved Union algorithm (Patrick, 2001). They suggested that all the positions corresponding to each “do not care” factor shall not be assigned until a coverage requirement which include that factor is dealt. With this perception, the improved Union algorithm (termed as ReqOrder) generated better results in term of test suite size reduction as compared with previous work. Despite of this, they also implemented input-output interaction testing by adopting in-parameter-order strategy (Yu and Kuo-Chung, 1998), which is known as ParaOrder (Wang et al., 2007; Ziyuan et al., 2008). For this strategy, an initial test suite will be constructed for a sub-system with small number of factors. The system is then extended by adding a new factor to get a test suite for the new sub-system. The extending process is repeated until all factors have been added into that system. As far as the input-output interaction test suite size is concerned, ParaOrder gave comparable results against Greedy algorithm and better test suites than Union algorithm and ReqOrder.

Meanwhile, Ziyuan et al. (2008) utilized one-test-at-a-time strategy (Cohen et al., 1997) and density concept (Patrick et al., 2002) to generate test suite. This approach is generally consumed longer computational time against others (Union, ReqOrder and ParaOrder) although it mostly produced better test suite than them. In addition, there is an interaction testing tool; test vector generator (TVG) (Arshem, 2009) which is capable to generate test suite based on input-output relationship, uniform strength and variable strength *t*-way coverage as well as random manner. However, its implementation details are unknown. While most of the strategies produce test cases in symbolic values, automatically maps these test cases back into actual data form is therefore another practical feature shall be taken into account. A recent strategy, GTWay (Zamli et al., 2011) started to address this automated mapping need. At a glance, GTWay implemented a preprocessing automated mapping system by employing Parser algorithm to capture the actual values from the fault file and map them into symbolic representations before they can be used for *t*-

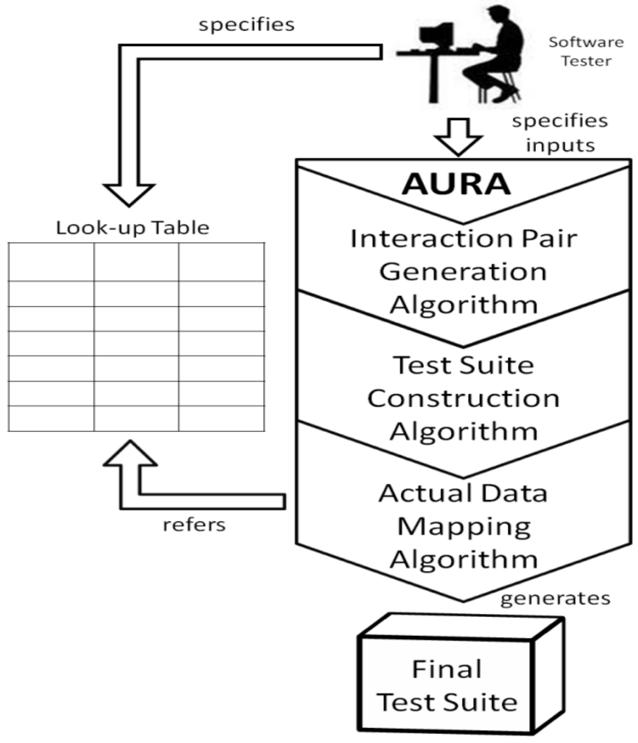
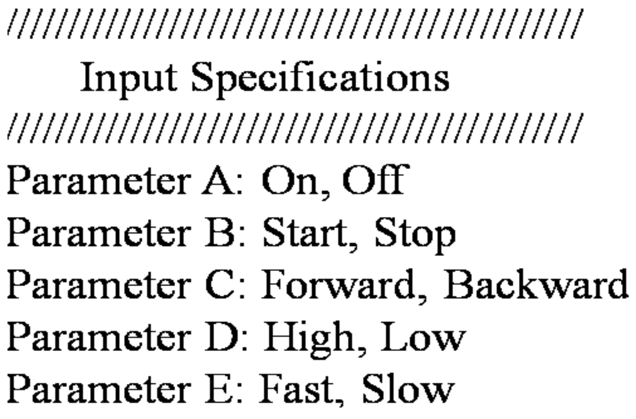

specifies
Software Tester
specifies inputs
Look-up Table
AURA
Interaction Pair Generation Algorithm
Test Suite Construction Algorithm
Actual Data Mapping Algorithm
refers
generates
Final Test Suite

Figure 1. Overview of AURA strategy.


//

Input Specifications

//

Parameter A: On, Off

Parameter B: Start, Stop

Parameter C: Forward, Backward

Parameter D: High, Low

Parameter E: Fast, Slow

Figure 2. A typical look-up table.

way test cases generation. Upon completion of test suite generation, GTWay remaps these symbolic data representations into actual data values. Nevertheless, GTWay consumed a significant portion of execution time to read and convert the actual data from the fault file and this lead to an overhead penalty incurred during the preprocessing mapping process.

OVERVIEW OF AURA STRATEGY

As input-output based relationship interaction testing is more

Table 1. Symbolic notations for inputs with 5 parameters (each 2 values).

Parameter	A	B	C	D	E
Base values	a1	b1	c1	d1	e1
	a2	b2	c2	d2	e2

flexible to accommodate the actual interactions for a typical SUT, we have opted to develop AURA strategy which supports input-output interaction test suite generation. Moreover, the urge of automated input-output mapping support in the sense of generating actual value test suites encourages AURA strategy to adopt this mapping automation feature as well. Throughout this section, the development of AURA strategy with input-output mapping supports will be highlighted accordingly. Basically, this strategy is composed of three algorithms: “interaction pair generation algorithm, test suite construction algorithm and actual data mapping algorithm”. The overview of the AURA strategy has been summarized in Figure 1. Referring back to Figure 1, a software tester clarifies and summarizes the inputs (in terms of actual data form) into a look-up table for typical software SUT. In order to generate intended test suite, the software tester then keys in the symbolic values of inputs into AURA strategy. Based on these inputs, ‘interaction pair generation’ algorithm will then be triggered to generate all possible interaction pairs, which are needed for successive operations.

After that, ‘test suite construction’ algorithm starts to construct test suite by exploiting the interaction pairs that are previously generated. Upon completion, the results are loaded into ‘actual data mapping’ algorithm for generating final test suite (in actual data form) as specified at the predefined look-up table file. Lastly, Figure 2 shows a typical look-up table that AURA strategy adopts.

Interaction pair generation algorithm

In AURA strategy, test suite generation is based on its corresponding interaction pairs. The interaction pairs are used to ensure the completeness of interaction coverage for specified coverage requirements. Hence, the details on generating the interaction pairs shall be discussed here. In order to generate interaction pairs, AURA strategy first needs to enumerate the corresponding parameter interactions groups. As mentioned earlier, AURA strategy is developed to support input-output based relationship, uniform strength and variable strength interaction testing. Therefore, one would ask how the AURA strategy enumerates the parameter interactions groups and also the interaction pairs with these different kinds of considerations. Indeed, AURA strategy generalizes and converts these parameter interactions groups into binary representations (that is, 0 and 1 s). After the conversion, for each parameter interactions group, 1 s (in binary form) is representing the parameters that involved in the parameter interactions whereas 0 s (in binary form) are the complementary parameters of 1 s. It is noted that parameter involves in parameter interactions is termed as interaction element throughout this paper. As illustration, consider the input parameters as shown in Figure 2. In fact, these inputs are summarized as symbolic data (Table 1) which AURA strategy adopts. Supposed that we intent to generate an input-output based relationship interaction test suite that comprised of 2 coverage requirements, *R* = (ABC, DE). As far as the parameter interactions groups are concerned, AURA strategy will generalize these as 11100 (for ABC) and 00011 (for DE) respectively. The similar mechanisms can be applied to both uniform and variable strength interaction testing as well. For instance, assumed that CA (2, 2^5) and VCA (2, 2^5 CA (3, 2^3)) in this case. For all of these, the parameter interactions groups

```
Algorithm of Interaction Pair Generation(factor set, F and interaction coverage requirements, R)
1: begin
2: let F = {$f_0$.....$f_{m-1}$} where F represents number of values defined for each factor, f and m =
   number of factor
3: let R = {$r_0$.....$r_j$} where R represent the set of interaction coverage requirements
4: initialize IE = { }, where IE represents interaction pairs partitioned base data structure
5: for each interaction coverage requirement, $r_i$
6: {
7:   for each combination of parameter interaction in $r_i$
8:   {
9:   if (factors are interaction elements)
10:      generate interaction pair for interaction elements;
11:   else
12:      append don't care ("X") values for non-interacting elements;
13:   store the interaction pair (with "X") into IE[i];
14: }
15:}
16: return IE;
17: end
```

Figure 3. Pseudo-code of interaction pair generation algorithm.

and their subsequent binary representations are summarized in Table 2.

Upon generation of aforementioned parameter interactions groups in binary representations, 'interaction pair generation' algorithm proceeds to generate all possible interaction pairs. To do so, for each parameter interactions group, exhaustive combinations will be formed within its interaction elements whereas the do not care ('X') values are assigned on the corresponding non interacting elements. Table 3 depicts the resultant interaction pairs for ABC and DE and Figure 3 shows the pseudo-code of 'interaction pair generation' algorithm. As could be seen in Figure 3, AURA strategy adopts dynamic partitioned base data structure, *IE*, in Interaction Pair Generation algorithm to hold the generated interaction pairs since it offers systematic and well organized space search rather than the unpartitioned base data structure, in which every search always begin from the first data of the bulky irregular data structure. This data structure is termed as dynamic since the number of partition for data structure relies on the number of parameter interactions group involved. In this example, AURA strategy used 2 partitioned data structure to store the resultant interaction pairs accordingly.

Test suite construction algorithm

First of all, AURA strategy is implemented based on one-test-at-a time basis (Ziyuan et al., 2008). In this basis, the test suite generation process begins with an empty test suite. Besides that, the interaction pairs sets that corresponding to a SUT input specifications are generated. Then, test cases are generated and added into that empty test suite one by one, until all interaction pairs are covered. Meanwhile, AURA strategy also gives non-deterministic output since the random selections are used to construct each test case. For this reason, this strategy not always produce similar test suite for every run; though, the generated test suite size is compromised. In addition, AURA strategy user can decide on the number of iterations, *n*. With this customizable looping system, if the user chooses a larger value of *n* for typical input, then there is higher chance for AURA strategy to give more optimized test suite. This is due to the fact that, by having more iterations (corresponding to higher value of *n*) on assigning additional set of random values and checking corresponding uncovered interaction pairs (in *IE*) process, AURA strategy gets higher possibility to obtain the test case which have more uncovered interaction pairs (in *IE*). Though, AURA strategy consumes extra computational time for these. In other side of coin, AURA strategy generates less optimized test suites with smaller *n* value but less execution time. With the aforementioned considerations, AURA strategy summons 'test suite construction' algorithm to generate test suite once the interaction pairs are generated. Figure 4 illustrates the pseudo-code of 'test suite construction' algorithm. In order to generate test suite, a test case, α is proposed by forming the first combination of exhaustive combinations (for interaction elements) and assigning random values to those non interacting elements from the first group of parameter interactions. As assigning random values are non-deterministic in nature, one may also get different kind of results for each proposed α. Then, based on *IE* as well, AURA strategy subsequently verifies the number of interaction pairs, β, which could be covered by α.

Next, α is promptly sent to final test suite, τ if only if it covered all of its corresponding interaction pairs (indicating an optimized test case has been found). Otherwise, AURA strategy will reassign another set of random values and check for its covering interaction pairs. As long as the optimized test case has not been discovered, AURA strategy will repeat the assigning and checking process for *n* (number of iterations) times. Among the *n* test cases been proposed, AURA strategy will select a test case greedily (that is, with the greatest number of uncovered interaction pairs in *IE*). The selected test case is then added into τ. The corresponding β in *IE* are then eliminated before the system proceeds to propose next α. The τ is considered completely formed as all interaction pairs in *IE* are covered, indicating all interaction coverage from coverage requirements are included.

Table 2. Summary of parameter interactions groups.

Types of interaction testing	Parameter interactions groups	Binary representations
Input-output based relationship	{ABC, DE}	{11100, 00011}
Uniform strength, CA (2, 2^5)	{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE}	{11000,10100, 10010, 10001, 01100, 01010, 01001, 00110, 00101, 00011}
Variable strength, VCA (2, 2^5 CA (3, 2^3)	{ AB, AC, AD, AE, BC, BD, BE, CD, CE, DE, CDE}	{11000,10100, 10010, 10001, 01100, 01010, 01001, 00110, 00101, 00011, 00111}

Table 3. The resultant interaction pairs for coverage requirements ABC and DE.

Parameter interaction	ABC					DE				
Parameter	A	B	C	D	E	A	B	C	D	E
Interaction pairs	a1	b1	c1	X	X	X	X	X	d1	e1
	a1	b1	c2	X	X	X	X	X	d1	e2
	a1	b2	c1	X	X	X	X	X	d2	e1
	a1	b2	c2	X	X	X	X	X	d2	e2
	a2	b1	c1	X	X					
	a2	b1	c2	X	X					
	a2	b2	c1	X	X					
	a2	b2	c2	X	X					

Actual data mapping algorithm

As existing strategies are focused on test suite minimization efforts; therefore, less intention has been drawn into the input-output mapping automation supports to map the test data from symbolic value to actual data form. Consequently, test engineers will have to manually perform the mapping process before the test suite can be used for testing execution phase or else some automated mapping efforts are still be required. In this scenario, AURA strategy has incorporated with this automated mapping concern as far as the test suite practicality is concerned. Specifically, the automated mapping support in AURA strategy is known as post-processing mapping system. In this case, the term of post-processing refers to the test cases generation process is prior to the automated input-output mapping process, which allows symbolic inputs to be executed on AURA strategy (instead of real values) and generates real value outputs via a predefined look-up table. It is noted that the look-up table is summarized by test engineers before they specify inputs (in symbolic values) to AURA strategy. Thus, this feature enhances the mapping automation support while maintaining the ease of data manipulation in test suite construction. In order to implement the post-processing automated input-output mapping system, AURA strategy employed Actual Data Mapping algorithm to support both symbolic values as well as actual data output generation. Figure 5 depicting the pseudo-code of 'actual data mapping' algorithm. In this case, AURA strategy generates symbolic values results which are similar output format as some other strategies possessed. Moreover, with this feature, AURA strategy is also capable to map back the test cases with their actual data based on the corresponding look-up table.

To accomplish this aim, the actual data information from the look-up table is stored in σ as shown in Figure 5. Then, for the first test case in the τ, AURA strategy maps that test case, which is in symbolic values back into actual data form based on σ. After that, the test case (in actual data form) is then written into an output file, 'ofile'. This mapping process is repeated until all test cases in the final test suite are converted. For instance illustration, Figure 6 shows a typical constructed test suite in terms of symbolic values and also the conversion from that to actual data based on our previous example.

RESULTS AND DISCUSSION

Here, we evaluate AURA strategy with current benchmarking input-output based relationship interaction testing strategies. As mentioned earlier, AURA strategy can support three distinctive types of interaction testing:

input-output based relationship interaction testing, uniform and variable strength interaction testing.

```
Algorithm of Test Suite Construction (number of iterations, n and interaction pairs, IE)
1: begin
2: initialize $\tau$ = { }, where $\tau$ represents final test suite
3: initialize $\alpha$ = { }, where $\alpha$ represents a test case that will be comprised of a set of
   factors
4: initialize $\beta$ = { }, where $\beta$ represents covered pairs corresponding to $\alpha$
5: for each interaction coverage requirement, $r_i$
6: {
7:    for each combination of parameter interaction in $r_i$
8:    {
9:      generate interaction pair for interaction elements;
10:     for n times of iterations
11:     {
12:        assign a random value for each non-interacting element;
13:       form $\alpha$;
14:       check $\beta$ on IE;
15:       if (maximum # of $\beta$)
16:          break;
17:       else
18:          update weight;
19:    }
20:    add $\alpha$ into $\tau$;
21:    remove $\beta$ in IE;
22:    if (IE is null)
23:       break;
24:}
25: if (IE is null)
26:    break;
27:}
28: return $\tau$
28:end
```

Figure 4. Pseudo-code of test suite construction algorithm.

```
Algorithm of Actual Data Mapping ($\tau$, output file name)
1:  begin
2:  define ofile as output file name
3:  initialize $\sigma$ = { } where $\sigma$ represents the list of actual data
4:    if (look-up table file is specified)
5:    {
6:       translate actual data from look-up table file to $\sigma$;
7:       for each test case in $\tau$
8:       {
9:          convert test case into actual data based on $\sigma$;
10:         write test case into ofile;
11:      }
12: }
13: else
12: {
13:    for each test case in $\tau$
14:       {write test case into ofile;}
15:}
16: end
```

Figure 5. Pseudo-code of actual data mapping algorithm.

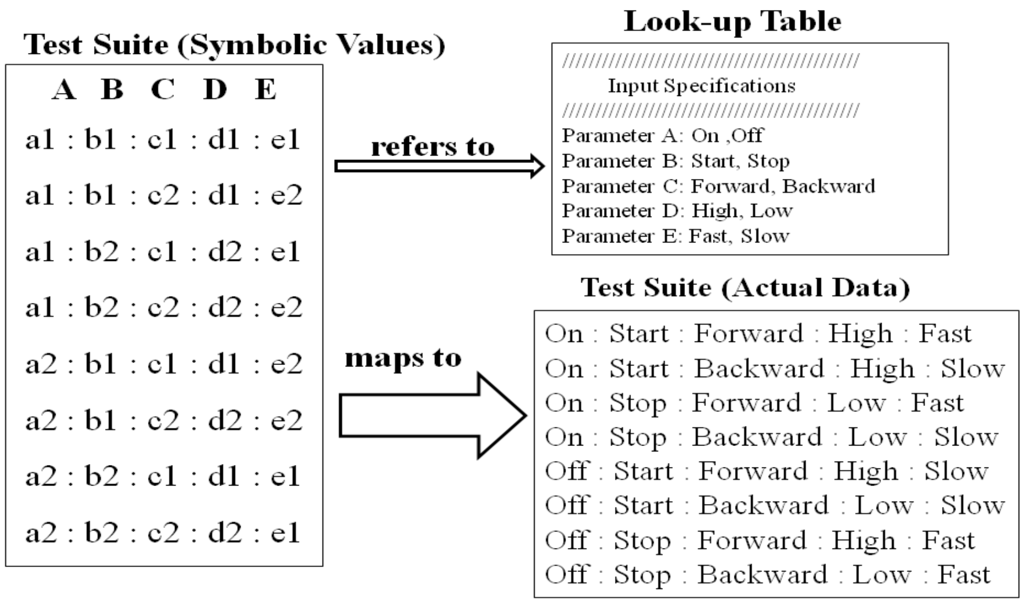


Figure 6. Conversion from symbolic values to actual data for a typical test suite.

Table 4. Comparison of different strategies for F1 = {3^{10}} and different size of R.

|R|	Density	ParaOrder	Union	TVG	AURA
10	86	105	503	86	89
20	95	103	858	105	99
30	116	117	1599	125	132
40	126	120	2057	135	139
50	135	148	2635	139	147
60	144	142	3257	150	158

Therefore, three experiments are conducted accordingly.For the first experiment, 2 set of factors are chosen to represent the systems with fixed-level factors and mixed-level factors respectively in order to evaluate the effectiveness of AURA strategy on input-output based relationship interaction testing. The fixed-level factors are consisted of 10 factors (parameters) where each factor have 3 levels (options) and defined as $F_1 = (3^{10})$ whereas mixed-level factors are 3 factors of 2 options, 3 factors of 3 options, 3 factors of 4 options and a factor with 5 options and denoted as $F_2 = (2^3 \times 3^3 \times 4^3 \times 5)$. AURA strategy is then required to generate test cases for these factor sets based on predefined input-output interactions. The input-output interactions are built by selecting coverage requirements from a pool of coverage requirements in Wang et al. (2007); they are also attached in appendix of this paper. Based on these, there are 6 iterations in the experiment for each factor set. The first 10 coverage requirements are included for the first iteration and defined as |R| = 10. The input-output interactions are then been added with another 10 following coverage requirements for consecutive iteration. Tables 4 and 5 have shown the generated test suites size from AURA strategy as well as other published strategies. Meanwhile, there are eight different input specifications (S_1 to S_8) of uniform strength interaction testing that commonly applied in other published strategies. Hence, AURA strategy is executed based on these and gave the test suite sizes as summarized in Table 6. As far as the benchmarking is concerned, existing variable strength interaction testing strategies tend to refer on the input specifications proposed by Cohen et al. (1997) and Myra et al. (2003b). Therefore, AURA strategy adopted these as the inputs for the last experiment.

The results are depicted in Table 7. Before any further discussion is made, several experimental clarifications are deduced here. First of all, it is noted that all the results are obtained by using Windows XP with a 2.80 GHz Core 2 Duo CPU and 2 GB RAM with Java (JDK 1.6) installed. Besides that, the data from others work (that is Density, ParaOrder, Union, TVG, PICT, AETG,

Table 5. Comparison of different strategies for F2 = {2^3 x 3^3 x 4^3 x 5} and different size of R.

|R|	Density	ParaOrder	Union	TVG	AURA
10	144	144	505	144	144
20	160	161	929	161	182
30	165	179	1861	179	200
40	165	183	2244	181	207
50	182	200	2820	194	222
60	197	204	3587	209	230

Table 6. Sizes of generated *t*-way combinatorial test suites (t = 3).

	Density	ParaOrder	TVG	PICT	AETG	GA	ACA	GA-N	IPO-N	IPO	Jenny	AURA
S_1	53	53	48	48	38	33	33	52	47	48	51	50
S_2	64	106	120	111	77	64	64	85	64	64	112	93
S_3	213	225	239	215	194	125	125	223	173	200	215	229
S_4	362	363	409	369	330	331	330	389	271	366	373	394
S_5	1592	1624	1949	1622	1473	1501	1496	1769	1102	1678	1572	1874
S_6	242	225	269	241	218	218	218	336	199	239	236	260
S_7	119	111	133	119	114	108	106	120	113	120	130	125
S_8	365	379	429	368	377	360	361	373	368	464	397	401

(S_1: 3^6; S_2: 4^6; S_3: 5^6; S_4: 6^6; S_5: 10^6; S_6: 5^7; S_7: $5^2 4^2 3^2$; S_8: $10^1 6^2 4^3 3^1$).

GA, ACA, GA-N, IPO-N, IPO, Jenny, SA and ACS) are collected from published articles in Myra et al. (2003b), Xiang et al. (2009) and Ziyuan et al. (2008). Since ACS strategy (Xiang et al., 2009) possesses randomness algorithm, and its results are obtained based on 200 iterations; hence, AURA strategy also performed 200 iterations throughout the experiments. As AURA strategy is non-deterministic in nature, 20 independent runs have been performed and the best results (with minimal test cases) of these runs are reported for each input. Lastly, no fair comparison for test cases generation time is able to be performed due to the differences or unspecified of computing environments in the published literatures. Thus, this aspect is excluded in our discussion. Based on Tables 4 and 5, 'density' produced smallest test suites for almost all inputs. However, AURA strategy also produced considerable results in this case. For instance, AURA strategy gave optimized test suite at |R| = 10 for F_2 = (2^3 x 3^3 x 4^3 x 5) and produced better results than ParaOrder as well as TVG in some other cases. Particularly, AURA strategy always generates smaller test suites than Union for each input. By referring to Table 6, none of the reported strategies is accomplished to give the best results for all input configurations of uniform strength interaction testing. In addition, we discovered that there is no big gap in terms of test suites sizes constructed by AURA strategy with some classic algorithms. For some inputs, AURA strategy even produced better output than Density, ParaOrder, PICT, GA-N, IPO and Jenny. Meanwhile, AURA strategy generated smaller size than TVG for most of the inputs as well. Despite of the test suite optimality has been concerned, most of the existing uniform strength interaction testing strategies (AETG, GA, ACA, GA-N, IPO-N, IPO and Jenny) do not address capability of input-output interaction testing as well as variable strength interaction testing.

Concerning variable strength interaction testing test suites as shown in Table 7, simulated annealing (SA) strategy is often effective in constructing small test suite size but it does not support input-output interaction testing. Besides that, it is worth to mention that AURA strategy frequently performed well against PICT as far as the size is concerned. In addition, AURA strategy is comparable in certain cases as compared to Density, ParaOrder and TVG, by generating smaller test suites. Moreover, there comprised of four input specifications at which AURA strategy produced optimized results that as optimized as SA strategy. They are namely: CA (3, 5^3), CA (3, 4^3) + CA (3, 5^3) and CA (3, $5^1 6^2$) from VCA [m; 2, $4^3 5^3 6^2$, (CA)] and also VCA (m; 2, $3^{20} 10^2$). Apart from supporting all forms of *t*-way testing possibilities (for example uniform strength, variable strength, and input-output based relation) and generating competitive test size, the main contribution of AURA strategy is the fact that it provides seamlessly integration of input-output mapping of actual data values as part of the strategy itself. Comparatively, AURA strategy can be put side-by-side with GTWay (Zamli et al., 2011). GTWay takes the preprocessing approach through the use of fault file to address its input-output mapping of actual data. Although

Table 7. Sizes of variable strength interaction testing test suites.

{C}	Density	ParaOrder	SA	PICT	TVG	ACS	AURA
VCA(m; 2, 3^{15}, {C})							
Ø	21	33	16	35	22	19	21
CA(3, 3^3)	28	27	27	81	27	27	28
CA(3, $3^3)^2$	28	33	27	729	30	27	30
CA(3, $3^3)^3$	28	33	27	785	30	27	31
CA(3, 3^4)	32	27	27	105	35	27	35
CA(3, 3^5)	40	45	33	131	41	38	42
CA(3, 3^4) CA(3, 3^5) CA(3, 3^6)	46	44	34	1376	53	40	50
CA(3, 3^6)	46	49	34	146	48	45	46
CA(3, 3^7)	53	54	41	154	54	48	53
CA(3, 3^9)	60	62	50	177	62	57	62
CA(3, 3^{15})	70	82	67	83	81	76	88
VCA(m; 2, $4^3 5^3 6^2$, {C})							
Ø	41	49	36	43	44	41	44
CA(3, 4^3)	64	64	64	384	67	64	64
CA(3, $4^3 5^2$)	131	141	100	781	132	104	127
CA(3, 5^3)	125	126	125	750	125	125	125
CA(3, 4^3) CA(3, 5^3)	125	129	125	8000	125	125	125
CA(3, $4^3 5^3 6^1$)	207	247	171	1266	237	201	224
CA(3, $5^1 6^2$)	180	180	180	900	180	180	180
CA(3, $4^3 5^3 6^2$)	256	307	214	261	302	255	287
VCA(m; 2, $3^{20} 10^2$, {C})							
Ø	100	100	100	100	101	100	100
CA(3, 3^{20})	100	103	100	940	103	100	107
CA(3, $3^{20} 10^2$)	401	442	304	423	423	396	586

useful, this preprocessing approach introduces some timing overhead to parse the actual parameter values and to directly manipulate actual data for test generation. Particularly, it is reported that the overhead penalty incurred in GTWay is directly proportional to the number of defined base test cases (approximately 50 us per additional defined test). Unlike GTWay, AURA strategy adopts post-processing approach, that is, the mapping to actual values is achieved after generation via look-up tables. Thus, no time is wasted on loading fault file and parsing parameter. In this case, AURA strategy provides better mapping automation feature than GTWay. Furthermore, unlike existing strategies, the iterations of assigning additional set of random values and checking corresponding uncovered interaction pairs in AURA strategy can be controlled (that is the number of iterations is customizable).

On one hand, when the optimality of test suite size is preferred than generation time, AURA strategy can increase the iterations looping number in order to generate more optimized test suite. On the other hand, AURA strategy can decrease the iterations looping to get fast test suite construction. This notable customization is useful to provide the flexibility for the users to decide their preference (either the optimality of test suite or fast generation time).

CONCLUSIONS

This paper has discussed the development of a flexible input-output based *t*-way strategy called, AURA. Experimental data shown AURA strategy generally produces competitive test suites for all these inputs. Moreover, AURA strategy that implemented the post-processing automation on converting back test suites into actual value form has alleviated the burden of software tester from mapping the test suites manually. In future work, more practical features such as seeding and constraint implementations shall be incorporated into AURA strategy to further enhance its reliability and usability.

ACKNOWLEDGMENTS

This research is partially funded by the generous fundamental grants: "investigating t-way test data reduction strategy using particle swarm optimization technique" from Ministry of Higher Education (MOHE) and the USM research university grants: "Development of variable strength interaction testing strategy for t-way test data generation" and PRGS: "implementing distributed t-way test data generator by using tuple space technology". The first author is the recipient of the USM postgraduate fellowship.

REFERENCES

Ahmed BS, Zamli KZ (2010). PSTG: A T-Way Strategy Adopting Particle Swarm Optimization. Fourth Asia Int. Conf. on Math./Anal. Model. & Comput. Simul. (AMS), pp. 1-5.
Ahmed BS, Zamli KZ, Lim CP (2011). Constructing a T-Way Interaction Test Suite Using Particle Swarm Optimization Approach. Int. J. Innov Comput. Inf. Control., 7(11): 1741-1758.
Arshem J (2009). Test Vector Generator Tool (TVG). Retrieved 20 December 2010, from http:///sourceforge.net/projects/tvg.
Baowen X, Lei X, Changhai N, William C, Chang CH (2003). Applying combinatorial method to test browser compatibility. Multimedia Softw. Eng., 2003. Proc. Fifth Int. Symp., pp.156-162.
Basili VR, Selby RW (1987). Comparing the Effectiveness of Software Testing Strategies. IEEE Trans. Softw Eng. SE-13(12): 1278-1296.
Beer A, Mohacsi S (2008). Efficient Test Data Generation for Variables with Complex Dependencies. 1st Int. Conf. on Softw. Test., Verification Valid, pp. 3-11.
Bryce RC, Colbourn CJ (2007). The density algorithm for pairwise interaction testing. Softw Test, Verification Reliab., 17(3): 159-182.
Cemal Y, Myra BC, Adam AP (2006). Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. IEEE Trans. Softw Eng. 32(1): 20-34.
Cheng C, Dumitrescu A, Schroeder P (2003). Generating small combinatorial test suites to cover input-output relationships. Proc. of third Int. Conf. on Qual. Softw., pp.76-82.
Cohen DM, Dalal SR, Fredman ML, Patton GC (1997). The AETG system: an approach to testing based on combinatorial design. IEEE Trans, Softw Eng., 23(7): 437-444.
Czerwonka J (2006). Pairwise Testing in Real World: Practical Extensions to Test Case Generators. Proc. of 24th Pac. Northwest Softw. Qual. Conf.
Jangbok K, Kyunghee C, Hoffman DM, Gihyun J (2007). White Box Pairwise Test Case Generation. Seventh Int. Conf. on Qual. Softw., QSIC, pp. 286-291.
Kimoto S, Tsuchiya T, Kikuno T (2008). Pairwise Testing in the Presence of Configuration Change Cost. Second Int. Conf. on Secur. Syst. Integr. & Reliab. Improv., SSIRI, pp. 32-38.
Klaib MFJ, Zamli KZ, Isa NAM, Younis MI, Abdullah R (2008). G2Way A Backtracking Strategy for Pairwise Test Data Generation. 15th Asia-Pac. Softw. Eng. Conf., APSEC, pp. 463-470.
Kuhn DR, Wallace DR, Gallo AM, Jr. (2004). Software fault interactions and implications for software testing. IEEE Trans. Softw Eng. 30(6): 418-421.
Lei Y, Kacker R, Kuhn DR, Okun V, Lawrence J (2007). IPOG: A General Strategy for T-Way Software Testing. 14th Annu. IEEE Int. Conf. & Workshops on the Eng. Comp.-Based Syst. ECBS, pp. 549-556.
Maity S, Nayak A (2005). Improved test generation algorithms for pair-wise testing. 16th IEEE Int. Symp. on Softw. Reliab. Eng. ISSRE., 10: 244.
McCaffrey JD (2009a). Generation of Pairwise Test Sets Using a Genetic Algorithm. Comp. Softw. & Appl. Conf., 2009. COMPSAC '09. 33rd Annu. IEEE Int., 626-631.
McCaffrey JD (2009b). Generation of pairwise test sets using a simulated bee colony algorithm. Inf. Reuse & Integr., 2009. IRI '09. IEEE Int. Conf.,pp.115-119.
Myra BC (2003). Augmenting Simulated Annealing to Build Interaction Test Suites. Int. Symp. on Softw. Reliab. Eng., pp. 394-394.
Myra BC, Peter BG, Warwick BM, Charles JC (2003a). Constructing test suites for interaction testing. Proc. of the 25th Int. Conf. on Softw. Eng. Portland, Oregon, IEEE Computer Society.
Myra BC, Peter BG, Warwick BM, Charles JC, James SC (2003b). Variable Strength Interaction Testing of Components. Proc. of the 27th Annu. Int. Conf. on Comp. Softw. & Appl., IEEE Computer Society.
Nie C, Leung H (2011). A survey of combinatorial testing. ACM Comput. Surv., 43(2): 1-29.
Patrick JS (2001). Black-box test reduction using input-output analysis. PhD dissertation, Illinois Institute of Technology, Chicago, IL, USA
Patrick JS, Bogdan K (2000). Black-box test reduction using input-output analysis. Proc. of the ACM SIGSOFT int. symp. on Softw. test. & anal. Portland, Oregon, United States, ACM.
Patrick JS, Pat F, Bogdan K (2002). Generating Expected Results for Automated Black-Box Testing. Proc. of the 17th IEEE int. conf. on Autom. softw. eng., IEEE Computer Society.
Reid SC (1997). An empirical analysis of equivalence partitioning, boundary value analysis and random testing. Proc. of Fourth Int. Softw. Metr. Symp., pp. 64-73.
Reussner R, Mayer J, Stafford J, Overhage S, Becker S, Schroeder P, Nie C, Xu B, Shi L, Dong G (2005). Automatic Test Generation for N-Way Combinatorial Testing. Qual. of Softw. Archit. and Softw. Qual., Springer Berlin / Heidelberg, 3712: 203-211.
Sunderam V, van Albada G, Sloot P, Dongarra J, Shi L, Nie C, Xu B (2005). A Software Debugging Method Based on Pairwise Testing. Comput. Sci. – ICCS 2005, Springer Berlin / Heidelberg, 3516: 55-81.
Wang Z, Nie C, Xu B (2007). Generating combinatorial test suite for interaction relationship. Fourth int. workshop on Softw. qual. assur.: in conjunction with the 6th ESEC/FSE jt. meet. Dubrovnik, Croatia, ACM.
Xiang C, Qing G, Ang L, Daoxu C (2009). Variable Strength Interaction Testing with an Ant Colony System Approach. Proc. of the 16th Asia-Pac. Softw. Eng. Conf., IEEE Computer Society.
Yan J, Jian Z (2006). Backtracking Algorithms and Search Heuristics to Generate Test Suites for Combinatorial Testing. 30th Annu. Int. Comput. Softw & Appl Conf, COMPSAC,385-394.
Yingxia C (2009). A New Strategy for Pairwise Test Case Generation. Workshop on Intel. Inf. Technol. Appl., 303-306.
Younis MI, Zamli KZ (2010). MC-MIPOG: A parallel t-way test generation strategy for multicore systems. ETRI J., 32(1): 73-83.
Yu L, Kuo-Chung T (1998). In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. The 3rd IEEE Int. Symp. on High-

Assur. Sys. Eng., IEEE Computer Society.

Yu L, Raghu K, Kuhn DR, Vadim O, James L (2008). IPOG-IPOG-D: efficient test generation for multi-way combinatorial testing. Softw Test Verif Reliab., 18(3): 125-148.

Zabil MHM, Zamli KZ, Othman RR (2011). On Sequence Based Interaction Testing. IEEE Symp. on Comput. and Inf., Kuala Lumpur, IEEE.,662-667.

Zamli KZ, Klaib MFJ, Younis MI, Isa NAM, Abdullah R (2011). Design and implementation of a t-way test data generation strategy with automated execution tool support. Inf Sci.

Zamli KZ, Younis MI (2010). Interaction Testing: From Pairwise to Variable Strength Interaction. Fourth Asia Int. Conf. on Math./Anal. Model. & Comput. Simul. (AMS), pp. 6-11.

Ziyuan W, Baowen X, Changhai N (2008). Greedy Heuristic Algorithms to Generate Variable Strength Combinatorial Test Suite. The Eighth Int. Conf. on Qual. Softw., QSIC., 155-160.

APPENDIX

The collection of coverage requirements in the experiment is shown as follows: there are 10 factors in both F_1 and F_2, which is $F = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}\}$. These 10 factors could be denoted by their corresponding sequence numbers and described as $F = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ for short (Wang et al., 2007).

Collection = {{1, 2, 7, 8}, {0, 1, 2, 9}, {4, 5, 7, 8}, {0, 1, 3, 9}, {0, 3, 8}, {6, 7, 8}, {4, 9}, {1, 3, 4}, {0, 2, 6, 7}, {4, 6}, {2, 3, 4, 8}, {2, 3, 5}, {5, 6}, {0, 6, 8}, {8, 9}, {0, 5}, {1, 3, 5, 9}, {1, 6, 7, 9}, {0, 4}, {0, 2, 3}, {1, 3, 6, 9}, {2, 4, 7, 8}, {0, 2, 6, 9}, {0, 1, 7, 8}, {0, 3, 7, 9}, {3, 4, 7, 8}, {1, 5, 7, 9}, {1, 3, 6, 8}, {1, 2, 5}, {3, 4, 5, 7}, {0, 2, 7, 9}, {1, 2, 3}, {1, 2, 6}, {2, 5, 9}, {3, 6, 7}, {1, 2, 4, 7}, {2, 5, 8}, {0, 1, 6, 7}, {3, 5, 8}, {0, 1, 2, 8}, {2, 3, 9}, {1, 5, 8}, {1, 3, 5, 7}, {0, 1, 2, 7}, {2, 4, 5, 7}, {1, 4, 5}, {0, 1, 7, 9}, {0, 1, 3, 6}, {1, 4, 8}, {3, 5, 7, 9}, {0, 6, 7, 9}, {2, 6, 7, 9}, {2, 6, 8}, {2, 3, 6}, {1, 3, 7, 9}, {2, 3, 7}, {0, 2, 7, 8}, {0, 1, 6, 9}, {1, 3, 7, 8}, {0, 1, 3, 7}}.