THE DESIGN AND IMPLEMENTATION OF A PAIRWISE STRATEGY SUPPORTING CONSTRAINTS AND SEEDING MECHANISM

Kamal Z. Zamli, Mohammed I Younis,
Ong Hui Yeh, Juliana Md Sharif

Software Engineering Group
School of Electrical and Electronic Engineering
Universiti Sains Malaysia Enginering Campus
14300 Nibong Tebal, Penang, Malaysia

Faculty of Electrical Engineering
Universiti Teknologi MARA,
13500 Permatang Pauh, Penang, Malaysia

Abstract

This paper describes the design and development of a pairwise test data generation, called 2TG, supporting seeding and constraints. In doing so, a number of experiments are discussed in order to prove the correctness of the implementation.

Keywords: *Pairwise Testing, Combinatorial Testing, Constraints and Seeding Support, Software and Hardware Testing.*

I. INTRODUCTION

Nowadays, human are increasingly dependent on software to assist as well as facilitate daily chores. In fact, whenever possible, most hardware implementation is now being substituted by its software counterpart. From the washing machine controllers, mobile phone applications to the sophisticated airplane control systems, the growing dependence 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 flexible and can be easily changed and customized as the need arises.

The continuous dependencies on software often raise dependability issues particularly when software is being employed on harsh and life threatening or (safety) critical applications. Here, rigorous software testing becomes very important. Many combinations of possible input parameters, hardware/software environments, and system conditions need to be tested and verified against the specification for conformance. Often, this results into combinatorial explosion problem (i.e. exorbitant number of test cases to consider for testing).

Combinatorial explosion problem poses one of the biggest challenges in modern computer science due to the fact that it often defies traditional approaches to analysis, verification, monitoring and control [1]. 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

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), the USM research university grants – "Development of Variable Strength Interaction Testing Strategy for T-Way Test Data Generation", and the USM short term grant –"Development of a Pairwise Test Data Generation Tool Wirh Seeding and Constraints Support".

due to the need for faster and higher capability processors along state-of-the-art computer hardware. Apart from parallel testing, systematic random testing 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. Any two combinations of parameter values are to be covered by at least one test. Because combinatorial explosion problem is NP-complete (i.e. no exact solution), 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) [1]. Furthermore, the size of the minimum pairwise test set also grows logarithmically with the number of parameters and quadratically with the number of values [2].

Apart from the aforementioned problems, the effectiveness of the test data generated can sometimes be questionable. In many cases, pairwise interaction is meaningless when more than one faulty input condition is introduced in the final generated test suite. Here, the problem could arise when one dominant faulty test input is accidentally masking other faults caused by other test inputs. In this case, it is likely that only the fault caused by that dominant faulty test input is detected, attended, and fixed accordingly. Thus, the faults caused by less dominant inputs are accidentally overlooked. In this manner, these overlooked faults can sometimes serve as hidden time bombs waiting to cause unwanted service disruptions and even system crash. To address this issue, there is a need for a constraint support mechanism at the algorithm level to ensure that such problem will never occur. Here, the constraint mechanism helps to enable test engineer to specify the unwanted (i.e. faulty and impossible) test cases from appearing in the final generated test suite. In this manner, other suitable test cases can be automatically selected as best fit alternatives.

Additionally, due to customer demands, there is often a set of pairwise combinations that are required to be as part of the final suite (i.e. as benchmarking test). To support this capability, the seeding support mechanism must also be in place as part of the implemented algorithm. In this case, unlike the constraint mechanism which prevents unwanted test cases to appear in the final test suite, the seeding mechanism ensures that (benchmark) test cases do appear even if they may not be the best fit values.

Addressing the aforementioned challenges, a new strategy has been proposed for pairwise testing that is not only efficient but also cater for constraints and seeding support mechanism. As part of the ongoing work [1-4], this paper describes the design and development of a pairwise test data generation, called 2TG, supporting seeding and constraints. In doing so, a number of experiments are discussed in order to prove the correctness of the implementation.

II. RELATED WORK

According to Yu et al [5], existing interaction strategies for pairwise testing can be categorized into two categories based on the dominant approaches, that is, algebraic approaches or computational approaches.

Algebraic approaches construct test sets using pre-defined rules or mathematical function [5]. 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 [5][6]. Orthogonal arrays (OA) [7] and covering arrays (CA) [8][9] are typical example of the strategies based on algebraic approach. Some variations of the algebraic approach also exploit recursion in order to permit the construction of larger test sets from smaller ones [10].

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) [11, 12] and its variant (AETGm) [13], employs a greedy algorithm to construct the test case, that is, each test covers as many uncovered combinations as possible. Because AETG uses random search algorithm, the generated test case is highly non-deterministic (i.e. the same input parameter model may lead to different test suites [14]). Other variants to AETG that use stochastic greedy algorithms are: GA (Generic Algorithm) and ACA (Ant Colony Algorithm) [8]. In some cases, they give optimal solution than original AETG, although they share the common characteristic as far as being non-deterministic in nature. In Parameter Order (IPO) strategy [15] builds a pairwise test set for the first two parameters. Then, IPO strategy extends the test set to cover the first three parameters, and continues to extend the test set until it builds a pairwise test set for all the parameters. In this manner, IPO generates the test case with greedy algorithms similar to AETG. Nevertheless, apart from deterministic in nature, covering one parameter at a time allows the IPO strategy to achieve a lower order of complexity than AETG.

Based on computational approach, Schroeder and Korel [14] developed a rather unique combinatorial strategy based on the input and output relationship. If one or more parameters are known to have insignificant effect on the system (i.e. don't care), then the strategy randomly selects the appropriate replacement of the don't care value in order to perform the reduction. Although useful for system with known input output relationship, no reduction is possible if all the parameters have the same importance.

A more recent strategies based on computational approaches are IRPS [16] GTWay[17], and AllPairs [18]. Like IPO, IRPS is deterministic in nature. Unlike IPO and other computational strategies, IRPS focuses on efficient data structure for storing and searching pairs. In this manner, IRPS appears to be the only strategy that supports higher order interactions of parameters (i.e. from pairwise up to 13 ways).

Like IRPS, GTWay strategy also targets for high order interaction of parameters. Unlike IRPS, GTWay strategy also supports automated execution of the generated test data.

Similar to IRPS, GTWay 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 [18].

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) [13]. 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 [13]. Unlike AETG, IPO, IRPS and All Pairs strategy, which builds a test set from scratch, heuristic search techniques can predict the known test set in advanced. However, there is no guarantee that the test set produced are the most optimum.

Concerning constraint and seeding mechanism, most existing strategy implementations (e.g. IRPS, GTWay, IPO, and Allpairs) have not appeared to consider the support as yet as evident by their implemented tools. For these reasons, we aim to investigate the constraints and seeding mechanism as part of our tool implementation of 2TG.

III. ILLUSTRATIVE EXAMPLE

To illustrate the concept of pairwise testing, consider an example of a pizza online ordering system as illustrated in Figure 1.

Figure 1. Pizza Online Ordering System

In this system, there are 3 options for the user to choose the crust, flavour and toppings. For each of the option, there are 2 selections available. For simplification, this pizza online ordering system options can be represented with integer numbers (see Figure 2).

Figure 2. Pizza Option Representation Using Integer

Here, the pizza option representation can also be translated into a table of 3 columns (or parameters) and 2 rows (or values).

Table 1. Pizza Ordering System with 3 parameters and 2 values

Base Values	Input Variables		
	A	B	C
	0	0	0
	1	1	1

Let the input variable be a set X = {A, B, C}. For simplicity, assume that the starting test case, termed base test case, has been identified (see Table 1). Here, integer values (e.g. 0, 1, 2…) are used in place of real data values to facilitate discussion.

In this case, exhaustive combinations would yield = 23 = 8 possible test combinations. Referring to Table 2, if parameter C is known to have insignificant effects on the system, then C input could be treated as don't care value. Thus, C could randomly take either 0 or 1 respectively. Based on this premise, one can select only an instance of each input combination to cover 2-way combination for AB at least once. In this case, there are two possible combinations for 2-way covering of AB. For instance, consider the input variable {0,0}. The first AB combination would be {0,0,0} and the second combination would be {0,0,1}. In order to cover for 2-way combination for AB, one can randomly select any one of the aforementioned combinations.

Table 2. Exhaustive Combinations

	Input Variables		
Base Values	A	B	C
	0	0	0
	1	1	1
Exhaustive Combinations	0	0	0
	0	0	1
	0	1	0
	0	1	1
	1	0	0
	1	0	1
	1	1	0
	1	1	1

Using this technique, the number of combinations can be reduced significantly. For instance, for 2-way combination AB, the total test data can be reduced to merely 4 (see Figure 3).

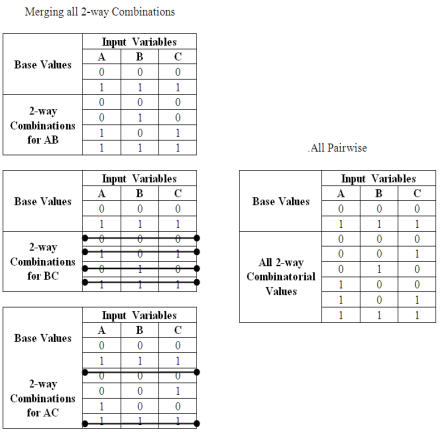

Figure 3. Merging of AB, BC, and AC

In reality, nevertheless, it is often difficult to establish for certain which variable has insignificant effect on the system. Thus, it is necessary to consider the impact of other 2-way combinations as well. In this example, there are 3 possibilities for 2-way interactions: AB, BC, and AC. Having considered AB and using similar approach given earlier, the values for the other 2-way combinations BC and AC can also be generated (see Figure 3).

Referring to Figure 3, an obvious observation is the fact that the total test data has been reduced from 8 exhaustively to 6 for pairwise, a reduction of 25%. The reduction technique or strategy illustrated here can be helpful as far as minimizing the required tests.

It can be noticed that there exists combinations of {0,0,1} corresponding to {Classic Hand Tossed, Vegetarian, Beef} and {1,0,1} corresponding to {Crunchy Thin, Vegetarian, Beef}. However, these combination are deemed illegal (i.e. as vegetarian implies no beef). Concerning seeding, they are the combinations that are required, necessary and desirable. As discussed earlier, the support for constraint and seeding are the scope of this work.

IV. PROPOSED STRATEGY, 2TG

In a nutshell, the 2TG strategy consists of seeding algorithm, the pairwise binary input combination algorithm and the constraint algorithm. The overview of 2TG strategy can be represented using the block diagram as illustrated in Figure 4.

The seeding algorithm involves capturing the specified test data directly into the final test suite as specified by the user. Binary input combination algorithm implements the interaction between parameters and generates pairwise combination accordingly. The constraint algorithm iteratively finds the test case that satisfies the constraints into the final test suite. The complete description for all algorithms can be seen in Figure 5,6,7 respectively.

Using our earlier example of a system with 3 parameters and 2 values, the

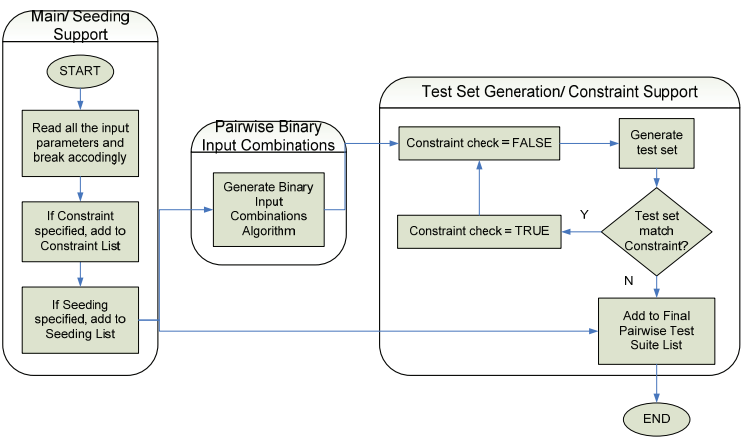

Figure 4. 2TG Strategy as Block Diagram

working of each algorithm will be illustrated in the next paragraph. As for the seeding algorithm, the combinations that are specified by the user are directly appended to the final test suite list.

```
Algorithm Seeding

1:    if seeding is specified
2:    begin
3:        read seeding parameters
4:        put in seeding list automatically
5:        add seeding to final test suite list
6:    end
```

Figure 5. Seeding Algorithm

```
Algorithm Pairwise Binary Input Combinations

1:   begin
2:   let limit =2^data length
3:   for  i=0 until reaches limit
4:       begin
5:            comb = convert integer to binary
6:           while length of comb< data.length
7:           comb ="0"+comb;
8:            initialise no_of_one
9:           for j=0 until j<comb.length
10:               if character j of comb='1' then
11:               increment no of one by 1
12:          if no of one equals to t value
13:          begin
14:              add comb to binary combination list
15:          end
16:      end
17: end
```

Figure 6.Binary Input Combination Algorithm

```
Algorithm Constraints

1:    begin
2:      for all combinations in binary combination list
3:        begin
4:          set constraint match to false
5:          generate pairwise test case by random generation
6:           if test case equals to constraint
7:                constraint match=true;
8:            if count iteration has reached limit  then break
9:            if constraint  match is false
10:              add test case into test suite list
11:          continue
12:       end
13:  end
```

Figure 7.Constraints Algorithm

Concerning the pairwise binary input combinations algorithm, the selection of don't care values to be randomized is highly dependent on the generated binary combination list. Here, the binary numbers representing the complete possible number of combination is first generated and the subsets with occurrences of two 1's are selected accordingly in the binary combination list. In our example, the three selected binary combinations are '011', '110' and '101'.

The constraints algorithm is slightly more complicated than both of the earlier given algorithms. Here, the constraints algorithm first checks the binary setting for the first column i.e. column A. Since the binary setting for the first column is '0', this represents the 'don't care' state. Table 3 shows the first pair of pairwise combinations for BC where the 'X' denotes the 'don't care' state for column A.

Table 3. First Pair of Pairwise Combinations for BC with Don't Care State

	Input Variables		
Base Values	**A**	**B**	**C**
	0	0	0
	1	1	1
First Pair	X	0	0

Therefore, a random number will be generated for column A. Assuming the random number generated by the algorithm is '0', the algorithm will store this random generated number inside column A. For the pairwise column (i.e. BC), the algorithm will generate an incrementing number starting from '0'. Therefore, the algorithm will now point to the next column i.e. column B. Now, since the binary setting for this column is '1', using recursive loop, the algorithm will generate '0' for this column. Next, pointing to column C, the algorithm will also generate a '0' for this column. Table 4 shows the first pair generated.

Now, since the first pair has been completely generated, the algorithm will check whether this pair matches the specified constraint parameter. Assuming the specified constraint parameter is {1,0,1}. Hence, the first generated pair does not match with the constraint parameter. Consequently, constraint match remains as FALSE. In the same manner, the constraint algorithm will continue to iterate and generate the next pairs. Table 5 shows the first and second pairs generated. From Table 5, the second pair generated is {1,0,1}. Again, the algorithm will check whether this pair matches with the specified constraint parameter i.e. {1,0,1}. Since now the match is found, the

constraint match becomes 'TRUE'.

Table 4. First Pair of Pairwise Combinations for BC

	Input Variables		
Base Values	**A**	**B**	**C**
	0	0	0
	1	1	1
First Pair	0	0	0

Table 5. Second Pair of Pairwise Combinations for BC

	Input Variables		
Base Values	**A**	**B**	**C**
	0	0	0
	1	1	1
First Pair	0	0	0
Second Pair	1	0	1

Next, the algorithm will check for value of limit. Here, the limit represents the maximum allowable loops set to prevent infinite loop in the case of non-feasible solution. In the case of no feasible solution, the algorithm will keep on looping until the limit is reached. If either the limit has been reached, the searching loop will terminate. If the constraint match is 'FALSE' then the pair will be added to the final test suite, otherwise, the algorithm will iterate further for an alternative pair. Table 6 shows the combinations for BC where the unwanted pair, {1,0,1} has been excluded from the final test suite list.

Table 6. Pairwise Combinations for BC with Constraint

	Input Variables		
Base Values	**A**	**B**	**C**
	0	0	0
	1	1	1
Exhaustive Pairwise Combinations for BC	0	0	0
	0	1	0
	1	1	1

The constraint algorithm will then iterate to the next combinations (i.e. AC and AB). The same iterations will be repeated until completion. Table 7 depicts one of the possible pairwise combinations.

Table 7. Pairwise Combinations with Constraints

	Input Variables		
Base Values	**A**	**B**	**C**
	0	0	0
	1	1	1
All Pairwise Combinatorial Values	0	0	0
	0	0	1
	0	1	0
	1	0	0
	1	1	1

V. DEMONSTRATION OF CORRECTNESS

In order to demonstrate the correctness of the 2TG strategy, 4 experiments have been conducted as follows:

- 2TG support for seeding
- 2TG support for constraints
- 2TG support for general pairwise generation with both seeding and constraints
- 2TG behaviour when there is no feasible solution possible

Here, the experiments are based on our implementation of 2TG using the Java programming language. Here, the 2TG takes three possible command line parameters as follows:

2TG –i {parameters} –c {constraints list} –s {seeding list}

Where

- I represents the input values
- c represents the constraints list
- s represents the seeding list

Here the {parameters} take a number separated by comma to represent parameters and values. For example, {3,3,4} represents a system with 2 3-valued and 1 4-valued parameters). The {constraints list} and {seeding lists} take a number separated by colon to represent the constraints and seeding. For instance, 1:0:1 represent a seeding (or constraints) of value 1 of parameter 0, value of parameter 0 of parameter 1, and value 1 of parameter 2.

A. 2TG Support for seeding

The objective of this experiment is to demonstrate the 2TG's support for seeding. In this experiment, the input parameter argument is {-i 3,3,3 -s 0:0:0,0:1:2} corresponding to 3 parameters with 3 values and seeding values of 0:0:0 and 0:1:2. It is expected that these two seeding input parameters will be part of the final suite list. Figure 8 depicts the output of this experiment.

As expected, the seeding parameters {0:0:0} and {0:1:2} are the first two pairs included in the Final Pairwise Test Suite List for both assessments. The result is a reduction of 33% of the total test data that is from 33=27 pairs exhaustively to 18 pairs.

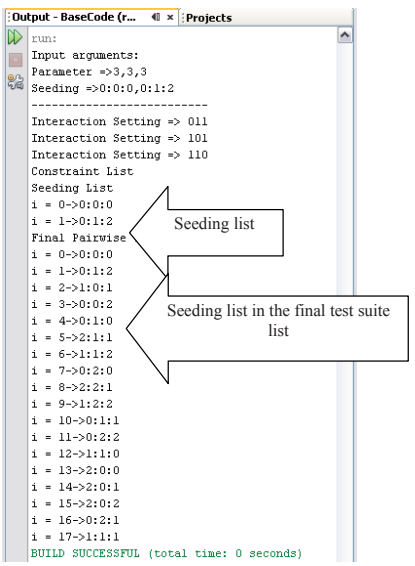

Figure 8. Output for Seeding

B. 2TG Support for constraints

The objective of this experiment is to demonstrate the 2TG's support for constraint. In this assessment, the input parameter is {–i 3, 3, 3 –c 0:0:0,0:1:2}. From the input parameters, there are 2 sets of constraints parameters; {0:0:0} and {0:1:2}. It is expected that the result will be smaller sized test suite where these two constraints will be excluded from the final test suite list. Figure 9 depicts the output of this experiment.

Figure 9 shows the final test suit list which consists of 14 pairs. The constraint parameters {0:0:0} and {0:1:2} have been excluded from the Final Pairwise Test Suite List. The result is a reduction of 48% of the total test data that is from 33=27 pairs exhaustively to 14 pairs only.

C. 2TG Support for general pairwise generation with both seeding and constraints

The objective of this experiment is to demonstrate the 2TG's support for general pairwise generation with both seeding and constraints. In this experiment, the input parameter is {-i 2,3,4 -c 0:0:0,0:1:2 -s 1:1:1,0:2:2}. It is expected that both seeding and constraint parameters given will be reflected in the final test suite list. It should be noted that the input parameters are non-uniform with 1 2-valued parameter, 1 3-valued parameter and 1 4-valued parameter. Figure 10 depicts the output of this experiment.

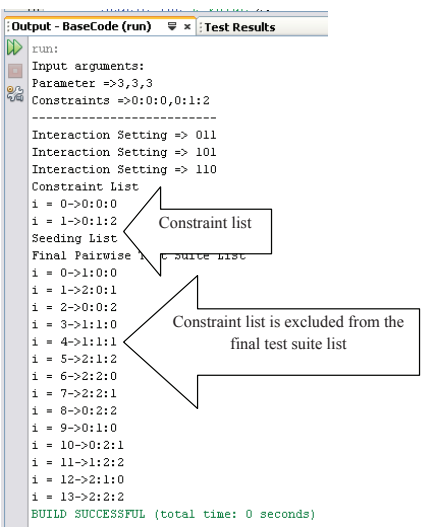

Figure 9. Output for Constraints

Referring to Figure 10, the final test suite includes two seeding parameters 0:0:0 and 0:1:2 whereas the constraint parameters 1:1:1 and 0:2:2 have been appropriately

excluded. There is a reduction of 25% of the total test data that is from 2x3x4 = 24 pairs to 18 pairs.

D. 2TG behaviour when there is no feasible solution possible

The objective of this experiment is to demonstrate the behaviour of the 2TG strategy when no feasible solution is possible.

In this experiment, the input variable is {-i 2,2,2 –c 0:0:0,0:0:1}. The constraint parameters chosen are {0:0:0} and {0:0:1}. In this case, since the base value is {2,2,2}, there are only two possible combinations for pairwise covering of AB. The first AB combination would be {0,0,0} and the second combination would be {0,0,1}. Since the constraint parameter set are {0:0:0} and {0:0:1}, therefore, there is no feasible solution for AB combinations. Thus, it is expected that none of the pairs covering AB combinations should appear. Figure 11 depicts the output of this experiment.

From the output in Figure 11, the final test suite covers only 6 pairs. As expected, none of the pairs that cover combination between AB appeared. For the result, there is a reduction of 25% of the total test data that is from 8 pairs to 6 pairs

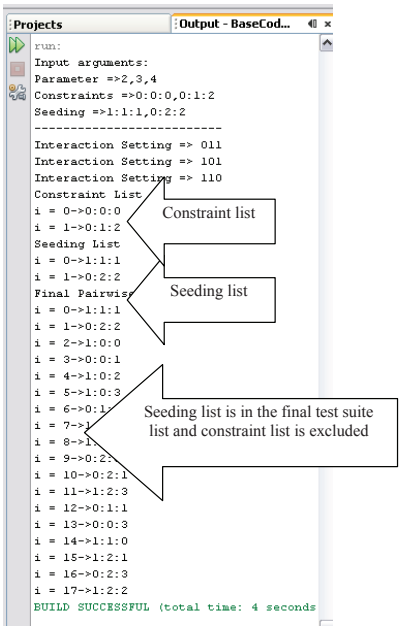

Figure 10. Output for Non-Uniform Parameter Values

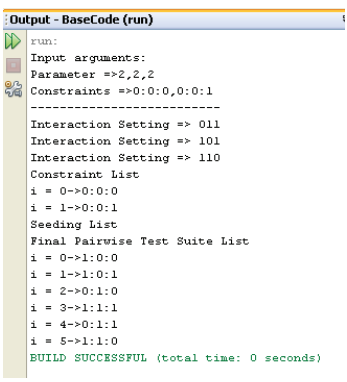

Figure 11. Output of Assessment for Non-Feasible Solution

VI. DISCUSSION AND CONCLUSION

The results obtained from experiment 1 demonstrate 2TG is able to support seeding, that is, by directly adding the specified seeding parameter into final test suite list. In the next assessment, the output showed that the constraint mechanism was fully supported by the strategy, where all the specified constraints

were not found in the final test suite list. Experiment 3 has demonstrated that 2TG strategy also supports non-uniform parameters values and finally experiment 4 has demonstrated the behaviour of the strategy when no feasible solution is available.

In a nut shell, the experimental results show that the developed strategy guarantees inclusion of the specified test cases by the seeding parameter as the first level of prioritized pair. By prioritizing the seeding parameter, none of the specified critical combinations will be missed out from the final generated test suite.

The constraint mechanism was proven to be working correctly. The strategy generates the pairs and checks against the specified unwanted combinations before adding to the final test suite. Often, this may result with smaller sized test suite or may be no feasible solution at all. In the case of no feasible solution, this strategy has been designed to check for matching pairs up to certain limit only. This approach prevents the problem of infinite loop.

In real life, parameter values may not always be uniform. Considering this reality, 2TG is also designed to support non-uniform parameter input values. Here, 2TG appears to work seamlessly well even in the presence of constraints and seeding requirements.

In short, the evaluation of 2TG has been promising. As part of future work, the pairwise support in 2TG will be extended to support higher order interaction along with the case study evaluations involving both hardware and software systems.

REFERENCES

[1] M.F.J. Klaib, "Development of An Automated Test Data Generation and Execution Strategy Using Combinatorial Approach", School of Electrical and Electronic Engineering, Universiti Sains Malaysia, PhD Thesis (2009).

[2] M.I. Younis, "Development of a Parallel T-Way Minimization Strategy for Combinatorial Testing", School of Electrical and Electronic Engineering, Universiti Sains Malaysia, PhD Thesis (2010).

[3] J. Md. Sharif, "Implementation of Seeding and Constraints Mechanism for Pairwise Test Data Generation", School of Electrical and Electronic Engineering, Universiti Sains Malaysia, MSc Dissertation (2010).

[4] K.Z.Zamli and M.I.Younis, "Interaction Testing: From Pairwise to Variable Strength Interaction", in Proc of the Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation (AMS2010), Kota Kinabalu, pp. 6-11.

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

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

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

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

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

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

[11] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C.Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design",IEEE Transactions On Software Engineering, 23(7), July 1997, pp. 437-444

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

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

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

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

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

[17] M.F.J. Klaib, K.Z. Zamli, N.A. Mat Isa, and R. Abdullah, "G2Way – A Backtracking Strategy for Pairwise Test Data Generation", in Proc. of the IEEE Asia Pacific Software Engineering Conference (APSEC 2008), Beijing, December 3-5, 2008

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