Int. J. Advance Soft Compu. Appl, Vol. 5, No. 3, December 2013
ISSN 2074-8523; Copyright © SCRG Publication, 2013

Implementing a *T*-Way Test Generation Strategy Using Bees Algorithm

Mohd Hazli Mohamed Zabil and Kamal Z. Zamli

College of Information Technology
Universiti Tenaga Nasional
e-mail: hazli@uniten.edu.my

Faculty of Computer Systems and Software Engineering
Universiti Malaysia Pahang
Pahang, Malaysia
e-mail: kamalz@ump.edu.my

Abstract

In order to ensure software performance as well as software quality, various testing techniques have been used to detect faults as early and as many as possible during the development phase. Over the last decade, the size and complexity of software developed have increased tremendously. Highly customizable software allow users to configure the software to the users' needs, however, if not tested adequately, these software are prone to interaction faults. This paper discusses our experiences on implementation of Bees algorithm for generating test cases to detect t-way interaction faults (where t signifies the interaction strength).

Keywords: *Interaction testing, software testing, t-way testing, Bees algorithm.*

1 Introduction

We use software in almost all activities in our life - performing our daily chores like washing, watching television, using smart phones and even while we are driving. With the advancement of technology, software not only resides in computers, but also in many embedded devices such as mobile phones and electronic appliances at home. To ensure acceptable quality, software needs to be tested before delivery. For highly configurable software, all of the possible combinations of configurations need to be tested accordingly to detect interaction faults. However, taken into consideration of time-to-market and resource constraints, performing exhaustive testing for all configured combinations is impractical and inefficient due to combinatorial explosion problem. For example,

let us take a system with 20 inputs with each input has 2 possible values. To perform exhaustive combinatorial testing, we need to execute 2^{20} (1,048,576) test cases, which is obviously impossible to test within finite time limit.

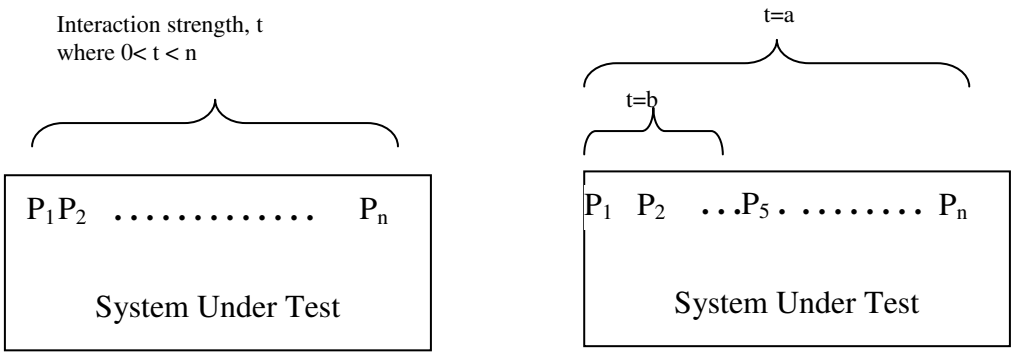
As a result, many *t*-way interaction testing strategy (where *t* indicates the interaction strength) have been proposed in the literature for the past 20 years. All of the strategies help in constructing test suites with minimal test cases to cover all intended interaction strength. Strategies such as GTWay [1], Automatic Efficient Test Generator (AETG) [2], Myra Implementation of AETG (mAETG) [3], In-Parameter Order (IPO) [4] , In-Parameter-Order General (IPOG) [5], Multi-Core IPOG (MC-IPOG) [6] and Jenny [7] generate test cases for uniform strength *t*-way (i.e. all parameters having interact uniformly among each others). In some cases, many researchs have demonstrated the need for generating variable-strength *t*-way combinations of inputs. As a result, newly developed *t*-way strategies have been proposed such as Simulated Annealing (SA) [8], Ant Colony System (ACS)[9] , Particle Swamp Optimization (PSO)[10] and Harmonic Search Strategy (HSS) [11] to consider the variable strength parameter interaction. The emergence of Artificial Intelligent (AI) based strategies has shown good results in term of producing optimal test case size. Based on such alluring prospects, we study Bees Algorithm as the basic for *t*-way test generation. Adopting other competing optimization algorithm other than SA, ACS, PSO and HSS might give rise to new perspectives on *t*-way test generation. In this paper, we discuss the implementation and the results of our experiment against other AI based *t*-way strategies.

The rest of this paper is organized as follows. Section 2 presents the theoretical background of *t*-way strategies. Section 3 introduces the Bees Algorithm, while section 4 discusses the implementation and experiment results. Lastly in section 5 we present our conclusion.

2 Theoretical Background

Over many years, sampling based strategies such as equivalence partitioning, boundary value analysis and decision table have been commonly used for test generation. Although helpful in detecting faults for some class of systems, these strategies do not detect faults due to interaction amongst input parameters. As a result, *t*-way strategies have been introduced. Here, *t* represents the interaction strength between parameters. In general, a uniform strength *t*-way constructs the basic of interaction testing. In uniform strength *t*-way, all of the parameters are tested to interact uniformly with *t* other parameters. For example, in a 2-way interaction test suite for a system with 5 input parameters, each of the parameter's value will have to be tested to interact with another parameter at least once. If we increase interaction strength to 3, that is, 3-way interaction, all parameters' value will be tested with another 2 parameters' value at least once.

Uniform strength produces decent test cases for interaction testing, however, in some cases, we might want to increase the interaction strength for a sub set of parameters in the system. In a testing scenario, a good tester knows, based on experience or requirement documents, that a particular interaction of parameter could give significant impact to the whole system, should failure occurs. Here, there is a need to rigorously test that particular sub set so as to increase the confidence level of the system, termed variable strength. As a result, on top of uniform strength, stronger interaction strength could be assign accordingly to a sub-parameter with higher risk of failure. For illustration, Fig.1(a) and Fig.1(b) below show the features of uniform strength and variable strength interaction respectively.



(a) Uniform interaction with strength *t* for *n* parameters

(b) Variable strength interaction, with t= a for all parameters (P_1 to P_n), and t=b for 5 sub parameters

Fig.1.Uniform and Variable strength interaction

For another illustration, consider a system under test (SUT) with 4 configurable parameters namely, A, B, C and D. Each parameter has 2 possible values as described in Table 1. To perform 2-way interaction test, all 2-way interactions for the 4 parameters A, B C and D have to be covered including that of {AB, AC, AD, BC, BD, CD}. The complete set of the interaction tuples are shown in Table 2.

As highlighted earlier, the *t*-way testing strategy addresses the issue of combinatorial explosion. Considering exhaustive generation of test cases for the said configuration, we consider all possible combination for each value of each parameter (see Table 3). Here, when the configuration increases, the size of the test case increase exponentially. For example, exhaustive test for a SUT with 5 parameters with 2 values for each parameter will give us $2^5 = 32$ test cases. For SUT with 10 parameters with 3 values for each parameter gives us $2^{10} = 59{,}049$ test cases. If we estimate that each test case needs even only 1 minute to execute, we need more than 41 days to perform such a test. For this reason, sampling

mechanism based on *t*-way testing is used to reduce the number of test cases, but at the same time give adequate coverage for the intended *t*-way interaction.

Table1: An example of a system with 4 parameters with 2 values

Parameters	**A**	**B**	**C**	**D**
Values	a1	b1	c1	d1
	a2	b2	c2	d2

Table 2: Complete interaction sets to be covered for 2-way testing

Parameters	**Interaction set**
AB	a1b1, a1b2,a2b1, a2b2
AC	a1c1, a1c2,a2c1,a2c2
AD	a1d1, a1d2,a2d1,a2d2
BC	b1c1, b1c2,b2c1,b2c2
BD	b1d1, b1d2,b2d1,b2d2
CD	c1d1, c1d2,c2d1,c2d2

Table 3: Exhaustive test suite

Test Case	**A**	**B**	**C**	**D**
1	a1	b1	c1	d1
2	a1	b1	c1	d2
3	a1	b1	c2	d1
4	a1	b1	c2	d2
5	a1	b2	c1	d1
6	a1	b2	c1	d2
7	a1	b2	c2	d1
8	a1	b2	c2	d2
9	a2	b1	c1	d1
10	a2	b1	c1	d2
11	a2	b1	c2	d1
12	a2	b1	c2	d2
13	a2	b2	c1	d1
14	a2	b2	c1	d2
15	a2	b2	c2	d1
16	a2	b2	c2	d2

At the end of test case generation, the final test suite, *F,* should cover every *t* combinations of parameter value configurations for at least once, the final test suite can be abstracted to covering array notation [4-5] shown in Equation 1.

$$F = CA(N, t, C) \quad (1)$$

where,

N = the number of test data inside the final test suite.

t = the interaction strength

C = value configuration can be represented as following:

$v_1^{p1}, v_2^{p2}, \ldots, v_n^{pn}$ which indicate that there are p_1 parameters with v_1 values, p_2 parameters with v_2 values and so on.

Using the aforementioned notation, a 2-way interaction for 4 input parameters with each parameter has 2 possible values can be written as CA(N, 2, 2^4). When a system consists of parameters with mixed number of values, we refer this type of covering array as mixed covering array (MCA). For example MCA(N, 2, 2^2 3^2 4^1) refers to a covering array of strength 2, with two 2-valued parameters, two 3-valued parameters and one 4-valued parameters.

With additional notation for variable strength, covering array notation for variable strength interaction, (VCA) can be denoted by

$$VCA(N, t, C, S) \quad (2)$$

where

N = the number of test data inside the final test suite.

t = the dominant interaction strength

C = value configuration can be represented as following:

$v_1^{p1}, v_2^{p2}, \ldots, v_n^{pn}$ which indicate that there are p1 parameters with v1 values, p2 parameters with v2 values and so on.

S = the multi-set of disjoint covering array with strength larger than t represented using notation similar to CA i.e. CA(N, t, C).

3 The Bees Algorithm (BA)

Bees Algorithm (BA) is a relatively new nature inspired, population based algorithm [12]. BA is developed based on the foraging behavior of honey bees. In order to forage food, a group of scout bee is sent to search the area around the

hive for flower patch. Bees can travel up to 10 kilometers in a day. The scout bees later will return to the hive and present its findings to the other bees (referred as unemployed bees) in a movement known as the waggle dance. By comparing the information gained in the waggles dance by scout bees, the unemployed bees will determine which flower patch has high quality. The quality of the nectar and its range from the hive are among factors considered to determine the chosen patches. More bees are sent to the more promising patches (high quality) and fewer bees are sent to other patches. The food level in the hive and the amount of nectar in the selected flower patches are monitored continuously and the information will be used in the next waggle dance. Inspired by the foraging behavior of honey bees, Bees Algorithm is developed. The basic form of Bees Algorithm is depicted in Fig. 2.

1. Initialize population with random solutions (*n*)
2. Evaluate fitness of the population
3. While (stopping criteria not met)
    4. Select (*m*) sites for neighborhood search (*ngh*)
    5. Recruit bees (*nep* and *nsp*) for selected site (*e* best elite sites, *m-e* non elite sites)
    6. Select the fittest bee from each patch
    7. Assign remaining bees to search randomly and evaluate their fitness
8. End While.

Fig.2: Bees Algorithm in its basic form.

In general, there are six parameters involved in performing BA, the initial population size *n*, the best solution for improvement *m*, elite solution *e*, number of bees for local search on elite solution *nep*, number of bees for local search on non elite solution *nsp* and the neighborhood size for local search *ngh*. The algorithm starts with selecting *n* random solutions. Then, the fitness of the initial solution is evaluated. From here, the best *m* solution is selected for neighborhood search (i.e. local search). From the best *m* number of solutions selected from *n*, the best *e* solution (elite solution) will be sent with *nep* bees. These *nep* bees, will try to improve and find a better solution around the current selected solution. To avoid local optima trap, the non-elite solution (*m-e*) will be sent with *nsp* bees. Similar to elite solution, *nsp* bees also try to improve the current solution, should there be a better solution. If a better solution if found by either *nep* or *nsp* bees, the better solution will replace the respective solution. In step 7, the solution is re-evaluated and sorted according to its fitness. The remaining bees in the population will randomly select a solution again and be sorted to get the best m solution. The process will be repeated until the criteria are met. In our case, the algorithm will keep looping until all of the intended interactions are covered.

Although BA is considered as a new swarm-based algorithm, BA has been used to solve many optimization problems. BA shows promising results in term of effectiveness, problem scale and performance published as in [13], [12], [14], [15], [16]. Although many of the BA implementation mentioned is functional optimization, BA is claimed suitable to solve not only functional, but also combinatorial optimization problems [12]. These factors motivate us to adopt BA further for *t*-way test data generation.

4 Implementation and Experiment Results

In order to generate *t*-way test cases, our strategy is divided into two main parts. The first part is the data part where all the information about the test case is stored. This includes the interaction set which stores all the interaction that needs to be covered and the test suite set (stores all the test case generated by BA). The interaction set is initialized at the beginning of the strategy. For experiment in this paper, the interaction set is generated using direct forward loop approach to generate all possible *t*-way interaction for the intended parameter-values. An example for generated interactions for 2-way, 4 parameters with each parameter having two values is shown in Table 2. The test case set is initialized as an empty set. Then, the test case set will store the best test case generated by BA in every cycle of the algorithm. At the end, the test set represents the best test suite for the intended configuration.

The second part is the optimization part. In this part, Bees Algorithm is implemented. At first, we initialized all the parameters needed for BA. Table IV shows the six parameters that are being used in our implementation. In the first step, a number of *n* test cases were generated randomly and its fitness sorted accordingly. In principle, the fitness of each test case is calculated by the ratio between number of interaction it covers with the number of maximum interaction can be covered. While there are still uncovered interactions, *m* best test cases from *n* which was generated randomly will be analyzed. For each test case in *m*, the algorithm tries to improve the test case by modify the test case to cover more interaction set via a process called neighborhood search. In the neighborhood search, *nep* improvement attempts will be performed to the elite test cases, while *nsp* improvement attempt will be performed to the non-elite test cases. If any improvement was made, the algorithm will update the fitness of the test case in the selected *m*. a new set of n-m test case then will be re-generated for the next cycle.

In each cycle, every time a test case is selected, the interaction covered by the respective test case will be removed from the interaction test set. At the same time, the selected test case will be added to the test suite set. Thus, the number of interaction set will be reduced in every cycle. The algorithm will keep running until the interaction set is empty (all interaction set has been covered).

Table 4: BA parameters

Parameter	Value	Description
n	20	Number of random test case generated
m	5	Selected test case with best coverage for neighborhood search
e	2	Elite test cases
nep	10	Number of improvement attempt for each elite test case
nsp	5	Number of improvement attempt for each non-elite site
ngh	0.5	Local search area size for nep and nsp

For the purpose of performance comparison, we benchmark our strategies with existing AI-based strategies. The results of existing AI-based strategies are obtained from the published works of the respective strategies. We have implemented our strategies in JAVA (Netbean 7.0) on a desktop PC running Windows XP SP3 with Intel Core2Duo and 2GB RAM.

We present our results in Table 5 and Table 6 below. For Table 5, we compare with other exiting AI-based strategies (HSS, GA, SA, ACA, PSTG), for low strength (2<t<3). For Table 6, we compare the results for CA(N, t,10,2) where *t* varied from 2 to 10 with existing strategies (AI and computational approaches). In the Tables 5 and 6, we have use Not Available (NA) to denote that there are no published result for the strategy and configuration of interest and Not Support (NS) to denote that the configuration is not supported by the given strategy.

Table 5: Comparison BA test suite against existing AI-based strategies

Configuration	BA	HSS	SA	GA	ACA	PSTG
CA(N, 2,4,3)	**9**	**9**	**9**	**9**	**9**	**9**
CA(N, 2,13,3)	19	18	**16**	17	17	17
CA(N, 2,10,10)	183	155	NA	157	159	NA
CA (N, 2, 10,5)	47	**43**	NA	NA	NA	45
CA(N, 3,6,3)	42	39	**33**	**33**	**33**	42
CA(N, 3,6,4)	108	70	**64**	**64**	**64**	102
CA(N, 3,6,5)	198	199	**152**	125	125	NA
CA(N, 3,7,5)	227	236	**201**	218	218	229

Table 6: CA(N, t, 10,2) with *t* varied from 2 to 10

t	BA	HSS	PSTG	IPOG	Jenny	TConfig
2	8	**7**	8	10	10	9
3	18	**16**	17	19	18	20
4	39	**37**	**37**	49	39	45
5	85	**81**	82	128	87	95
6	162	**158**	**158**	352	169	183
7	**298**	**298**	NS	NS	311	NS
8	503	498	NS	NS	521	NS
9	545	512	NS	NS	788	NS
10	**1024**	**1024**	NS	NS	**1024**	NS

Referring to Table 5 and Table 6, although not the best, BA produces comparable results against other strategies. In Table IV, the size of test suites produce is larger as compared to the other AI-based strategies. We believe the results can be improved upon fine-tuning the parameters in BA. In Table VI, BA manages to get 2 optimum results which are comparable to that of HSS and Jenny.

5 Conclusion

In this paper, we have discussed our preliminary works on adopting Bees Algorithm to generate interaction testing test data. We discuss the design of the algorithm and share our experiment results. As part of our future work, we are still optimizing our strategies to demonstrate the effectiveness with known case study, support variable strength interaction as well as to incorporate seeding and constraints while generating *t*-way test data.

ACKNOWLEDGEMENTS

This paper is partly funded by Universiti Tenaga Nasional Training Grant, the UMP RDU Short Term Grants: "Development of Test Generation, Execution, and Coverage Tools for Postgraduate & Undergraduate Teaching of Software Testing" and the MOE ERGS Grant: "CSTWay: A Computational Strategy for Sequence Based T-Way Testing".

References

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

[2] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. 1994. The Automatic Efficient Test Generator (AETG) System, *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pp. 303 –309.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. 1997. The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437 –444.

[4] Y. Lei and K.-C. Tai.1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing in *Proceedings of the 3rd IEEE International Symposium on High-Assurance Systems Engineering*, pp. 254–261.

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

[6] M.I. Younid and K.Z. Zamli. 2010. MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems, *ETRI Journal*, vol. 32. Taejon, COREE, REPUBLIQUE DE: Electronics and Telecommunications Research Institute.

[7]Bob Jenkin. 2012. "Jenny" [Online]. Available: http://burtleburtle.net/bob/math/jenny.html. [Accessed: 21-Jun-2012].

[8] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. 2003. Augmenting Simulated Annealing to Build Interaction Test Suites, *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pp. 394-405.

[9] X. Chen, Q. Gu, A. Li, and D. Chen. 2009. Variable Strength Interaction Testing with an Ant Colony System Approach, *Proceedings of the Asia Pacific Software Engineering Conference,* pp. 160 –167.

[10]B. S. Ahmed, K. Z. Zamli, and C. P. Lim. 2011. Constructing a T-Way Interaction Test Suite Using the Particle Swarm Optimization Approach, *International Journal of Innovative Computing and Information Control*, vol. 8, no. 1, pp. 1–10.

[11]A. R. A. Alsewari and K. Z. Zamli. 2012. Design and Implementation of a Harmony-Search-Based Variable-Strength t-way Testing Strategy with Constraints Support, *Information Software Technolology*, vol. 54, no. 6, pp. 553–568.

[12]D.T. Pham, A. Ghanbarzadeh, E.Koc, S.Otri, S.Rahim, and M.Zaidi. 2006. The Bees Algorithm - A Novel Tool for Complex Optimization Problems, *Proceedings of Innovative Production Machines and Systems Virtual Conference,* pp. 454-461.

[13]S. Anantasate and P. Bhasaputra. 2011. A Multi-Objective Bees Algorithm for Multi-Objective Optimal Power Flow Problem, *Proceedings of the 8th*

International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, pp. 852 –856.

[14]D. T. Pham, M. Castellani, and A. A. Fahmy. 2008. Learning the Inverse Kinematics of a Robot Manipulator using the Bees Algorithm, *Proceedings of the 6th IEEE International Conference on* I*ndustrial Informatics,* pp. 493 – 498.

[15]D. T. Pham, S. Otri, A. Ghanbarzadeh, and E. Koc. 2006. Application of the Bees Algorithm to the Training of Learning Vector Quantisation Networks for Control Chart Pattern Recognition, *Proceedings of the Information and Communication Technologies,* vol. 1, pp. 1624 –1629.

[16]D. T. Pham and M. Kalyoncu. 2009. Optimisation of a Fuzzy Logic Controller for a Flexible Single-link Robot Arm using the Bees Algorithm, *Proceedings of 7th IEEE International Conference on Industrial Informatics*, pp. 475 –480.