Journal homepage www.jzs.univsul.edu.iq
Journal of Zankoy Sulaimani
Part-A- (Pure and Applied Sciences)

Generating Pairwise Combinatorial Interaction Test Suites Using Single Objective Dragonfly Optimisation Algorithm

Bestoun S. Ahmed[1]

1 Software and Informatics Engineering Department, Engineering College, Salahaddin University – Erbil
E-mail: bestoun.ahmed@su.edu.krd

Article info

Original: 8 June 2016
Revised: 30 July 2016
Accepted: 16 October 2016
Published online: 20 March 2017

Key Words:
Combinatorial interaction testing; Software testing; Test generation tools; Dragonfly optimisation; Search-based software engineering; Test case design techniques.

Abstract

Combinatorial interaction testing has been addressed as an important and effective software testing technique recently. The technique can significantly reduce the number of test cases. It provides an alternative method to exhaustive testing and design of experiments (DOE) by allowing a minimised set of tests to represent the actual set of test cases. The reduction of the test cases is based on the combination of input parameters for the system-under-test. This combination could be considered as input-configuration of different software families. Pairwise combinatorial test suite takes the interaction of two input parameters into consideration instead of many parameter interactions. Evidences showed that this test suite can detect most of the faults in the software-under-test as compared to higher interactions. This paper shows the adaptation and assessment of Dragon Fly (DF), a novel swarm intelligent optimisation algorithm, for pairwise combinatorial test generation. The design of the algorithm is addressed in the paper. The algorithm is evaluated extensively through different experiments and benchmarks. The evaluation shows the efficiency of the proposed technique for test suite generation and the usefulness of DF optimisation algorithm for future investigations.

Introduction

Software quality assurance is one of the emerging important issues in the last decade. It aims to deliver software with minimum defects in which meets specific levels of functionality according to reliability and performance [1]. In addition, it is the systematic pattern of all actions for providing and proving the ability of a software process to build good products [2]. It also tries to improve the development process from the requirement step until the end.

With the vastly growing of software systems and their configurations, there is a chance for fault due to the combinations of these configurations especially for high configurable software systems. While the traditional test design techniques are useful for fault discovering and prevention, they may not be adequate to take care of faults due to combinations of input components and configurations [3]. Considering all the combinations of configuration leads to exhaustive testing which is impossible due to time and resource constraints [4-6]. For this reason, there is a need to minimise the number of test cases by designing effective test cases that have the same impact as exhaustive testing.

Many sampling strategies have been proposed in the literature. For example, equivalent partitioning, partitions the equivalent inputs of the system into groups [7]. The aim here is to design a test suite that covers all of these groups by taking at least one test for each of them. Another example of the sampling

strategy is the boundary value where the test suite is designed to cover the boundaries of the input values of the system. Combinatorial interaction testing is another sampling and test design technique that aims to design a test suite to cover all of the combinations among the input parameters of a software system.

Evidences in the literature showed that most of the faults (due to the interaction) in software systems could be detected with pairwise interaction, i.e., when the interaction is between two parameters of the input [8, 9]. Hence, the pairwise strategies aim is to construct pairwise test suites by searching and covering all of the interactions among the input parameters at least once. This could be a difficult task in case of highly configurable systems, which leads to combinatorial explosion and non-deterministic polynomial-time hard (NP-hard) problems [10]. To this end, different meta-heuristic algorithms have been used to construct pairwise test suites, including Particle Swarm Optimisation (PSO) [11], Genetic Algorithms (GA), [12] Ant Colony Algorithm (ACA) [12], and Tabu Search [13].

Dragonfly algorithm (DFA) [14] is a novel swarm intelligence optimisation technique that has been proposed recently. The technique produced promising results as compared to other state of the art techniques and introduces a new algorithm to solve evolutionary style problems. In line with the Search-based Software Engineering (SBSE) practices [15], this paper assess the use of DFA for pairwise test suite generation. The contribution of the work is mainly twofold: first the paper tries to asses and figure out how DFA benefits the test generation strategy; second, it also shows the design, adaptation and implementation of this algorithm for test design techniques.

The rest of this paper is organised as follows. Section 2 presents the mathematical notations and objects of combinatorial interaction testing. Section 3 summarises the related works in the literature. Section 4 shows the concepts of DFA and discusses its features. Section 5 discusses the design concepts of the technique, an adaptation of the algorithm and the implementation. Section 6 contains the results of the evaluation process. Finally, Section 7 concludes the paper.

Covering Array (Ca)

Covering array (CA) is a combinatorial mathematical object that represents the optimised set of combinatorial test suites. The object assumes that all interactions among the features are occurring within one array. Mathematically, CA is defined as bellow [16]:

Definition 1 : *In its general form, $CA_\lambda(N ; t, k, v)$ is an $N \times k$ array over $(0, \ldots, v-1)$ such that every $B=\{b_0, ..., b_{d-1}\} \in$ is λ-covered and every $N \times d$ sub-array contains all ordered subsets from v features of size t at least λ times, where the set of column $B=\{b_0, ..., b_{d-1}\} \supseteq \{0, ..., k-1\}$.*

As far as we are looking for the optimal array, the value of $\lambda=1$ which means that all the t-tuples of the input functions occur at least once and thus the notation becomes $CA(N;t,k,v)$ [17]. However, this notation assumes that the number of features in each function is equal, which is not the case in real applications. Here, the interaction strength t =2 since the pairwise test suite is considered in this paper. Normally each function has a different number of features depending on the system-under-test. To this end, mixed covering array (MCA) is used as a practical alternative, which is defined as below [18]:

Definition 2 : *MCA $(N; t, k, (v_1, v_2, \ldots v_k))$, is an $N \times k$ array on v levels, where the rows of each $N \times t$ sub-array cover and all t interactions of values from the t columns occur at least once .*

For more flexibility in the notation, the array can be presented by MCA $(N;d, v^k)$ and can be used for a fixed-level CA, such as CA $(N;t, v^k)$ [18]. So far, these notations have been used within combinatorial testing literature.

Related Works

Significant efforts in the literature have focused on combinatorial interaction testing. Much more efforts have focused on pairwise testing since it can detect faults effectively in many practical efforts. Basically, pairwise strategies are trying to construct pairwise test suites by searching for more pairwise interactions to cover within an optimised set. Using this approach, different strategies and algorithms have been

implemented in the literature (e.g., [9, 19, 20]). Owing to the complexity of the search space and the problem nature, searching for optimum pairwise set is considered as an NP-hard problem [21]. To this end, different strategies implemented in the literature to construct near optimum solution. In general, there are two approaches for this construction. Either the strategy follow an algebraic or computational approached for construction [22].

Algebraic approach follows some mathematical formulations and instructions to construct the pairwise test cases. For example, it may follow the Orthogonal Array (OA) construction to form the final test suite [23]. This approach may produce optimum results for small size problems, however it cannot construct test suites for large problem. The approach has a lack of constructing test suite when there are different number of values between the input parameters of the software-under-test. Computational approach in another hand is following some form of computation to construct the pairwise test suites. This approach is more suitable to construct test cases for real applications since it is more flexible with the number of parameters and values. Those strategies following this approach need to all pair interaction possibilities then checking the coverage of each proposed test case with the generated pair interactions. In such a checking process, there is significant searching efforts required in the combinatorial space in order to generate the required test suite until all pair interactions have been covered. There are two efforts for constructing test cases in this approach: one-test-at-a-time or one-parameter-at-a-time [22]. In case of one-test-at-a-time, the algorithm generates one test case and checks its coverage with the pair interactions. Automatic Efficient Test Generator (AETG) [24] and its variant (mAETG) [25], are well-known examples in this approach. In contrast, strategies with one-parameter-at-a-time fashion construct the test case by adding one parameter at a time and check for its coverage with the pair interactions. In Parameter Order (IPO) [26] is a well-known example in this approach.

Evidences in the literature also showed that with the help of Artificial Intelligence (AI) and optimisation methods, we can develop more flexible and optimum strategies. Two of the most common implemented artificial intelligence based strategies are based on Genetic Algorithm (GA) [12] and Ant Colony Algorithm (ACA) [12]. Despite giving good results both strategies appear to suffer from heavyweight computational process rendering difficulties to support high order parameters and values. For this, the search for other lightweight artificial intelligence strategies is an essential demand. We have also implemented Particle Swarm Optimisation (PSO) [27] for test generation and it could generate efficient test cases.

Nowadays, there are different new optimisation algorithms developed and come out in the literature. These algorithms need to be assessed for different problems to know the features and drawbacks for each of them. Some of these algorithms have been assessed already (e.g., [28, 29]). Dragonfly algorithm (DFA) is one of the new and novel algorithm developed recently that shows impressive results for few problems [14]. This paper shows the adaptation of this algorithm to generate pairwise test suites.

Dragonfly Optimisation

Dragonfly algorithm (DFA) is a novel meta-heuristic algorithm that has been proposed recently by Mirjalili [14]. The algorithm inspired by the natural dynamic and static behaviour of dragonflies swarming. The dynamic swarming behaviour of the dragonfly is inspired by the migration behaviour and the static is inspired by the hunting behaviour. The former called the “migratory” and the latter called “feeding” swarm. The swarming behaviour plays the main role in the inspiration of the algorithm. The static and dynamic behaviour of dragonfly are similar to exploration and exploitation in the meta-heuristic optimisation algorithms. To explore different area in the search space, the dragonflies divide themselves to sub-swarms, which is the static swarm that plays the role of exploration phase. These small groups of dragonflies are flying back and forth in a small area to hunt flying butterflies and mosquitoes. In the dynamic fly, the dragonflies are flying in bigger swarms and in one direction to play the role of exploitation phase [14]. These behaviours of the dragonflies are flowed three main principles with the neighbourhood which are: separation, alignment, and cohesion [30]. Separation refers to the collision avoidance of individuals from others in the neighbourhood; alignment arranges the velocity matching among individuals in the neighbourhood while the

cohesion deals with the tendency of these individuals towards the centre of mass in the neighbourhood. Figure 1 shows these components of the algorithm with the inspiration of dragonfly behaviour.

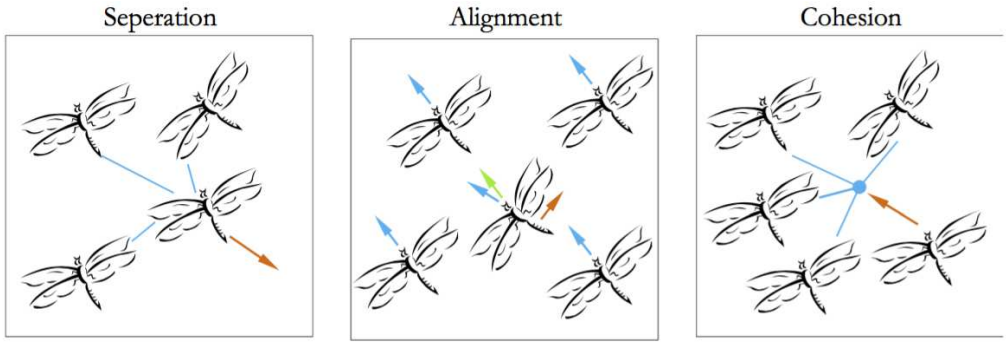

Figure 1. The separation, alignment, and cohesion components of DFA

Since the objective of the swarm is to survive, hence the survival flies toward food source and tries to distract outward enemies [14]. These other two components of DFA are shown in Figure 2.

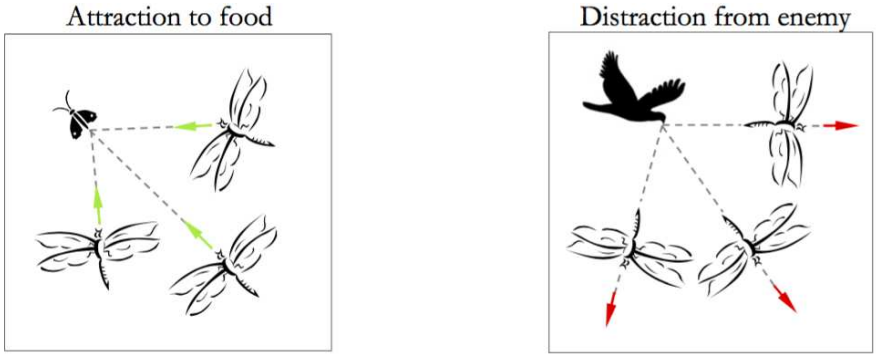

Figure 2. The attraction and distraction of DFA

Mathematically, these five components are represented and modelled as follows. The separation is calculated base on eq.1 [14].

$$S_i = -\sum_{j=1}^{N} X - X_j \dots\dots\dots\dots\dots\dots\dots\dots\dots (1)$$

where the number of neighbouring individuals denoted as N, current position of the individual is denoted by X, while X_j shows the position of the j^{th} neighbouring individual. The alignment component is calculated base on eq.2 .

$$A_i = \frac{\sum_{j=1}^{N} V_j}{N} \dots\dots\dots\dots\dots\dots\dots\dots (2)$$

where the velocity of the j^{th} neighbour is denoted as V_j. The cohesion is calculated using eq. 3

$$C_i = \frac{\sum_{j=1}^{N} X_j}{N} - X \dots\dots\dots\dots\dots\dots (3)$$

where the current position of the individual is denoted by X, the number of neighbourhoods are denoted as N, and the position of the j^{th} neighbour is denoted by X_j . The attraction equation can be seen in eq. 4:

$$F_i = X^+ - X \dots\dots\dots\dots\dots (4)$$

where X^+ is the food source position and X is the current position of the individual. Finally, the distraction of the enemies can be modelled as in eq.5.

$$E_i = X^- - X \quad \ldots\ldots\ldots\ldots\ldots (5)$$

where X^- is the enemy position and X is the current position of the individual.

The update role of the algorithm works in a systematic and artificial way by considering two update vectors, Step vector (ΔX) and Position vector (X). DX has an important role in the movement direction of the dragonflies and it is defined in eq.6:

$$\Delta X_{t+1} = (sS_i + aA_i + cC_i + fF_i + eE_i) + w\Delta X_i \ldots\ldots\ldots\ldots\ldots (6)$$

where t is the iteration counter, (s) is the separation weight, (S_i) is the separation of i^{th} individual, (a) is the alignment weight and (A) is the alignment of individual, (c) is the cohesion weight and (C_i) is the cohesion of the i^{th} individual, (f) is the food factor and (F_i) is the food source of the i^{th} individual, (e) is the enemy factor and (E_i) is the enemy position of the individual, and finally (w) is the inertia weight. These factors need accurate running for implementation. In this research we follow the literature [14] to set these parameters, (i.e., w = 0.9–0.2, s = 0.1, a = 0.1, c = 0.7, f = 1, e = 1)

As far as this step vector is calculated by eq.6, the vectors' positions are calculated by eq.7 as follows:

$$X_{t+1} = X_t + \Delta X_{t+1} \quad \ldots\ldots\ldots\ldots\ldots\ldots (7)$$

DFA takes the neighbourhood as 2D (circle), 3D (sphere), or nD (hyper-sphere) dimensions of the neighbour dragonflies. This condition is applicable when there is at least one neighbour. However, when the dragonfly cannot find this neighbour, it tends to take a random walk using the Le´vy flight mechanism. Here the position is updated using the Levy update equations [29, 31] as in eq.8 :

$$X_{t+1} = X_t + Le'vy(d) \times X_t \ldots\ldots\ldots\ldots\ldots (8)$$

where (t) is the current iteration, and d is the dimension of the position vectors. Le´vy flight itself can be obtained in the above equation by eq.9 as follows:

$$Le'vy(x) = 0.01 \times \frac{r_1 \times \delta}{|r_2|^{1/\beta}} \ldots\ldots\ldots\ldots\ldots (9)$$

where r_1, r_2 are random numbers between [0,1], β is a contact variable equals to 1.5. The δ factor in eq.9 is calculated by eq. 10 as follows:

$$\delta = \left(\frac{\Gamma(1+\beta) \times \sin\left(\frac{\Pi\beta}{2}\right)}{\Gamma\left(\frac{1+\beta}{2}\right) \times \beta \times 2^{\left(\frac{\beta-1}{2}\right)}} \right)^{1/\beta} \ldots\ldots\ldots\ldots\ldots (10)$$

where Γ(x) = (x-1)!

Base on the aforementioned equations and arrangements of the DFA, the Pseudo-code of the algorithm is summarised in Algorithm 1.

Algorithm 1: DFA general algorithm

```
Initialize the dragonflies population X_i(i = 1, 2, ..., n)
Initialize step vectors ΔX_i(i = 1, 2, ..., n)
while the end condition is not satisfied do
    Calculate the objective values of all dragonflies
    Update the food source and enemy
    Update w, s, a, c, f, and e
    Calculate S, A, C, F, and E using eq.1 to eq.5
    Update neighbouring radius
    if a dragonfly has at least one neighbouring dragonfly then
        Update velocity vector using eq.6
        Update position vector using eq.7
    else
        Update position vector using eq.8
    Check and correct the new positions based on the boundaries of
    variables
```

Algorithm Adaptation and Implementation

The DFA is adapted in this research for pairwise test cases generation to assess its optimisation efficiency for test generation. Here, the aim is to generation a CA where the interaction strength t=2. The objective function is to cover all pairwise interaction of input parameters at least once. To this end, another algorithm is used to generate the pairwise interactions first. The algorithm generates these interactions in an iterative way and then put them in a list. This list is used later by the DFA to computer the objective function. Algorithm 2 shows the Pseudo-code of the DFA for pairwise test suite generation.

The DFA starts by taking the input parameters and values. In addition, it takes the pairwise interactions list (2-tuples) that must be covered totally. The output of the algorithm is a pairwise test suite that covers the 2-tuples. The algorithm initialises a random dragonfly swarm base on the *k* columns and filled with *v* values. Hence, the algorithm iterates until the 2-tuples list gets empty. The algorithm evaluates each dragonfly base on its coverage of interactions, then it chooses the best one (dBest). The algorithm then updates the swarm base on the equations 1 to 5. Now, the algorithm evaluates the neighbour dragonflies and updates the velocity and positions base on equations 6 and 7. However, if there is no neighbour dragonfly, the algorithm uses the levy equations for update as in equation 8. Finally, the best dragonfly is chosen and its corresponding interactions are removed from the 2-tuples.

Algorithm 2: Dragonfly pairwise generation algorithm

Input: Input parameters k and values v, all pairwise combinations list $2-tuples$

Output: A pairwise test suite

```
Generate an initial dragonfly swarm base in k and v
Iter ← 1
while 2 − tples ≠ empty do
    while Iter < Max. Iter do
        foreach dragonfly d do
            check coverage with 2 − tuples find best best dragonfly "dBest"
        Update the food source and enemy
        Update w, s, a, c, f, and e
        Calculate S, A, C, F, and E using eq.1 to eq.5
        Update neighbouring radius
        if a dragonfly has at least one neighbouring dragonfly then
            Update velocity vector using eq.6
            Update position vector using eq.7
        else
            Update position vector using eq.8
    if best coverage is met by X_{t+1} then
        dBest ← dBest_{t+1}
    Add dBest to the test suite
    Remove all the related combinations from the t − tuples list
```

Evaluation Experiments

The evaluation and benchmarking focuses on the efficiency of the generation for DFA as well as the comparison of the algorithm with otters tools and counterpart algorithms. As far as the efficiency is considered, the size of the final generated test suites (i.e., number of rows) is reported for each algorithm and tool. Here, the benchmarks are adopted from evidence in the literature [11, 20, 24]. The results are compared with the published results for Genetic Algorithm (GA), Ant Colony Algorithm (ACA), and PSO which are heuristic and meta-heuristic counterpart algorithms. In addition the results are also compared with, Jenny [32] IPOG [33], AETG [24] and mAETG [34] which are computational tools. The algorithms are implemented and executed on our environment which consist of Laptop PC with MacOS X El Capitan 10.11.5, 2.9 GHz Intel Core i5, 8GB of DDR3. Tables 1, 2 reported these results clearly. It should be mentioned that cells marked NA (not available) indicate that the results are unavailable from the literature.

Table-1: Comparison of DFA with Existing Published Results

Configurations	AETG	mAETG	GA	ACA	IPOG	Jenny	TConfig	PICT	PSO	DFA
CA (N;2, 3^3)	NA	NA	NA	NA	11	9	10	10	9	9
CA (N;2, 3^4)	9	9	9	9	12	13	10	13	9	9
CA (N;2, 3^{13})	15	17	17	17	12	20	20	20	17	17
CA (N;2, 5^{15})	NA	NA	NA	NA	50	45	48	47	45	45
CA (N;2, 10^{15})	NA	NA	157	159	176	157	170	170	170	168
MCA (N;2, 5^1 3^3 2^2)	19	20	15	16	19	41	22	21	21	21
MCA (N;2, 6^1 5^1 4^4 3^4 2^2)	34	35	33	32	36	31	33	38	39	37
MCA (N;2, 7^1 6^1 5^1 4^4 3^4 2^2)	45	44	42	42	44	51	49	46	49	48
MCA(N;2, $10^1$$9^1$$8^4$$7^4$$6^5$$5^4$$4^3$$3^2$$2^1$	NA	NA	NA	NA	91	98	92	101	97	98

Table-2: 10 Parameters Variable Values Comparison of DFA with Existing Tools

V	IPOG	Jenny	TConfig	PICT	PSO	DFA
3	20	19	17	18	17	17
4	31	30	31	31	29	30
5	50	45	48	47	45	45
6	68	62	64	66	62	61
7	90	83	85	88	81	83
8	117	104	114	112	109	107
9	142	129	139	139	139	141
10	176	157	170	170	170	170

Tables 1 and 2 show the best reported results for 10 runs of the DFA as well as the comparison between the algorithm and other tools and algorithms. Clearly from the tables we can see that DFA generates test suites with satisfactory results as compared to other results. However, as compared to PSO, it generates better results in some cases and equivalent results in others. Overall, in Table 1, GA and ACA generate better results. However, this result is due to the compaction method used to compress the results produced by those algorithms. Hence, we cannot consider those published results as the actual results of the GA and ACA. IPOG, Jenny, PICT, and TConfig are able to produce impressive results (See Table 2). However, these strategies are designed for specific purposes. For example, IPOG is designed to generate test suites with many parameters and variables and it generates “good enough” in most of the cases. Jenny, PICT and TConfig are designed to generate test suites faster with. Taking DFA, it can generate test suites efficiently. However, it needs more investigation in case of higher interaction strengths.

Conclusions

This paper discusses the adaptation, implementation, and evaluation of the new dragonfly algorithm (DFA) for the use of pairwise test suite generation. The algorithm is implemented successfully for this purpose. Experimental evaluation is conducted for the algorithm to evaluate its efficiency against its counterpart PSO algorithm as well as other pairwise tools. The algorithm shows impressive results against other algorithms. The algorithm also showed that it can generate and optimise test cases. As compared to the state of the art algorithm, DFA shows better results in some benchmarks. However, more evaluation and experiments needed to figure out clearly its efficiency. This can be observed clearly by implementing the algorithm for higher combination strengths in the future.

References

[1] R. S. Pressman and B. R. Maxim, *"Software Engineering: A Practitioner's Approach*", 8th edition ed.: McGraw-Hill, (2015).

[2] L. Baresi and M. Pezzè, *"An introduction to software testing"*, Electronic Notes in Theoretical Computer Science, Vol. 148, pp. 89-111. (2006).

[3] B. S. Ahmed, M. A. Sahib, and M. Y. Potrus, *"Generating combinatorial test cases using Simplified Swarm Optimization (SSO) algorithm for automated GUI functional testing"*, Engineering Science and Technology, an International Journal, Vol. 17, pp. 218-226. (2014).

[4] D. Hoskins, R. C. Turban, and C. J. Colbourn, *"Experimental designs in software engineering: d-optimal designs and covering arrays"*, in ACM Workshop on Interdisciplinary Software Engineering Research, Newport Beach, CA, USA, pp. 55 - 66. (2004),

[5] W. Afzal, R. Torkar, and R. Feldt, *"A systematic review of search-based testing for non-functional system properties"*, Information and Software Technology, vol. 51, pp. 957-976. (2009).

[6] T. Mahmoud and B. S. Ahmed, *"An efficient strategy for covering array construction with fuzzy logic-*

based adaptive swarm optimization for software testing use", Expert Systems with Applications, Vol. 42, pp. 8753-8765. (2015).

[7] B. B. Agarwal, S. P. Tayal, and M. Gupta, *Software engineering and testing*. Hingham; Toronto: Infinity Science Press, (2010).

[8] J. Czerwonka, *"Pairwise testing in real world: practical extensions to test case generator,"* in 24th Pacific Northwest Software Quality Conference, Portland, Oregon, USA, pp. 419-430.(2006).

[9] J. Czerwonka. *Pairwise independent combinatorial testing (PICT) download page*. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/hh439673(v=vs.85).aspx (may 2008)

[10]Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "*IPOG-IPOG-D: efficient test generation for multi-way combinatorial testing,"* Software Testing, Verification & Reliability, Vol. 18, pp. 125-148. (2008).

[11]B. S. Ahmed and K. Z. Zamli, *"The Development of a Particle Swarm Based Optimization Strategy for Pairwise Testing,"* Journal of Artificial Intelligence, Vol. 4, pp. 156-165. (2011).

[12]T. Shiba, T. Tsuchiya, and T. Kikuno, *"Using artificial life techniques to generate test cases for combinatorial testing,"* in 28th Annual International Computer Software and Applications Conference, Hong Kong, Vol. 1, pp. 72-77. (2004).

[13]K. J. Nurmela, *"Upper bounds for covering arrays by tabu search,"* Discrete Applied Mathematics, Vol. 138, pp. 143-152. (2004).

[14]S. Mirjalili, *"Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems,"* Neural Computing and Applications, Vol. 27, pp. 1053-1073. (2016).

[15]M. Harman, *"The current state and future of search based software engineering,"* presented at the 2007 Future of Software Engineering, (2007).

[16]C. J. Colbourn, G. Kéri, P. P. R. Soriano, and J.-C. Schlage-Puchta, *"Covering and radius-covering arrays: Constructions and classification," Discrete* Applied Mathematics, Vol. 158, pp. 1158-1180. (2010).

[17]A. Hartman and L. Raskin, *"Problems and algorithms for covering arrays,"* Discrete Mathematics, Vol. 284, pp. 149-156. (2004).

[18]L. Gonzalez-Hernandez, *"New bounds for mixed covering arrays in t-way testing with uniform strength,"* Information and Software Technology, Vol. 59, pp. 17-32. (2015).

[19]R. C. Bryce, C. J. Colbourn, and M. B. Cohen, *"A framework of greedy methods for constructing interaction test suites,"* in 27th International Conference on Software Engineering, Oxford, UK, pp. 146-155.(2005).

[20] M. F. J. Klaib, K. Z. Zamli, N. A. M. Isa, M. I. Younis, and R. Abdullah, "*G2Way a backtracking strategy for pairwise test data generation,"* in 5th Asia-Pacific Software Engineering Conference, Beijing, Chaina, pp. 463 - 470.(2008).

[21] A. Calvagna and A. Gargantini, *"IPO-s: incremental generation of combinatorial interaction test data based on symmetries of covering arrays,"* in IEEE International Conference on Software Testing, Verification, and Validation Workshops, Denver, Colorado, USA, pp. 10-18. (2009).

[22] C. Nie and H. Leung, *"A survey of combinatorial testing,"* ACM Computing Surveys, Vol. 43, pp. 1-29. (2011).

[23] C. S. Cheng, *"Orthogonal arrays with variable numbers of symbols,"* The Annals of Statistics, Vol. 8, pp. 447-453. (1980).

[24] 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, Vol. 23, pp. 437-444. (1997).

[25] B. Garvin, M. Cohen, and M. Dwyer, *"Evaluating improvements to a meta-heuristic search for constrained interaction testing," Empirical Software Engineering*, Vol. 16, pp. 61-102, 2011/02/01 2011.

[26] K. C. Tai and Y. Lie, *"In-parameter-order: a test generation strategy for pairwise testing*," in 3rd

IEEE International Symposium on High-Assurance Systems Engineering, Washington, DC , USA, pp. 254-261. (1998).

[27] B. S. Ahmed and K. Z. Zamli, *"A greedy Particle Swarm Optimization strategy for t-way software testing*," in The Electrical and Electronic Postgraduate Colloquium (EEPC2011), Dusun Eco Resort, Bentong, Pahang, Malaysia, (2011).

[28] A. B. Nasser, F. Hujainah, A. A. Alsewari, and K. Z. Zamli, *"Sequence and sequence-less T-way test suite generation strategy based on flower pollination algorithm,"* in 2015 IEEE Student Conference on Research and Development (SCOReD), pp. 676-680. (2015).

[29] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, *"Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm,"* Information and Software Technology, Vol. 66, pp. 13-29. (2015).

[30] C. W. Reynolds, *"Flocks, herds and schools: A distributed behavioral model*," SIGGRAPH Comput. Graph., Vol. 21, pp. 25-34. (1987).

[31] X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms:* Second Edition: Luniver Press, (2010).

[32] M. Grindal, J. Offutt, and S. F. Andler, *"Combination testing strategies: a survey*," Software Testing, Verification & Reliability, Vol. 15, pp. 167 – 199. (2005).

[33] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, *"IPOG: a general strategy for t-way software testing,"* in 4th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson, Arizona, pp. 549-556.(2007).

[34] C. Yilmaz, M. B. Cohen, and A. Porter, *"Covering arrays for efficient fault characterization in complex configuration spaces,"* in 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, pp. 45-54. (2004).