Information and Software Technology 66 (2015) 13–29

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm

Bestoun S. Ahmed [a,*], Taib Sh. Abdulsamad [b], Moayad Y. Potrus [a]

[a] Software Engineering Department, Engineering College, Salahaddin University-Hawler (SUH), 44002, Erbil – Kurdistan
[b] Statistic & Computer Department, College Of Commerce, University of Sulaimani, SulaimaniNwe Street 27, Zone 209, Sulaimania, Kurdistan

ARTICLE INFO

Article history:
Received 17 January 2015
Received in revised form 16 May 2015
Accepted 16 May 2015
Available online 21 May 2015

Keywords:
Combinatorial testing
Search-based software testing
Cuckoo Search
Covering array
Test generation tools
Mutation testing

ABSTRACT

Context: Software has become an innovative solution nowadays for many applications and methods in science and engineering. Ensuring the quality and correctness of software is challenging because each program has different configurations and input domains. To ensure the quality of software, all possible configurations and input combinations need to be evaluated against their expected outputs. However, this exhaustive test is impractical because of time and resource constraints due to the large domain of input and configurations. Thus, different sampling techniques have been used to sample these input domains and configurations.
Objective: Combinatorial testing can be used to effectively detect faults in software-under-test. This technique uses combinatorial optimization concepts to systematically minimize the number of test cases by considering the combinations of inputs. This paper proposes a new strategy to generate combinatorial test suite by using Cuckoo Search concepts.
Method: Cuckoo Search is used in the design and implementation of a strategy to construct optimized combinatorial sets. The strategy consists of different algorithms for construction. These algorithms are combined to serve the Cuckoo Search.
Results: The efficiency and performance of the new technique were proven through different experiment sets. The effectiveness of the strategy is assessed by applying the generated test suites on a real-world case study for the purpose of functional testing.
Conclusion: Results show that the generated test suites can detect faults effectively. In addition, the strategy also opens a new direction for the application of Cuckoo Search in the context of software engineering.

1. Introduction

Testing is the process of evaluating the functionality of a system to identify any gaps, errors, missing requirements, and other features. This process ensures the sound operation of software [1]. In general, testing is mainly classified as either functional and structural [2,3]. The former method is referred to as "black box testing," and the latter is called "white box testing" [2–4].

In functional testing, the tester ignores the internal structure of the system-under-test and focuses only on the inputs and expected outputs. The technique serves the overall functionality validation of the system, thereby identifying both valid and invalid inputs from the customer's point of view. Structural testing is used to detect logical errors in software [3]. The tester needs to gather information on the internal structure of the system-under-test and to use information with regard to the data structures and algorithms surrounded by the code [5].

Unlike in structural testing, creating a data set (i.e., test data generation) is an important task in functional testing because of the lack of information about the internal design. Previous studies have reported many test data generation methods. In general, these methods use the available information in software requirement specifications, which provide knowledge about input requirements. The tester considers all possible input domains when selecting test cases for the software-under-test. However, considering all inputs is impossible in many practical applications because of time and resource constraints. Hence, the role of test design techniques is highly important.

* Corresponding author.
E-mail addresses: bestoon82@gmail.com (B.S. Ahmed), Taib.shamsadin@yahoo.com (T.Sh. Abdulsamad), moayad_75@yahoo.com (M.Y. Potrus).

http://dx.doi.org/10.1016/j.infsof.2015.05.005
0950-5849/

A test design technique is used to systematically select test cases through a specific sampling mechanism. This procedure optimizes the number of test cases to obtain an optimum test suite, thereby eliminating the time and cost of the testing phase in software development. Different studies proposed various functional test design techniques, such as equivalence class partitioning, boundary value analysis, and cause and effect analysis via decision tables [3,6]. In general, the tester aims to use more than one testing method because different faults may be detected when different testing methods are used. However, with the vast growth and development of software systems and their configurations, the probability of the occurrence of faults has increased because of the combinations of these configurations, particularly for highly configurable software systems. Traditional test design techniques are useful for fault discovery and prevention. However, such techniques cannot detect faults that are caused by the combinations of input components and configurations [7]. Considering all configuration combinations leads to exhaustive testing, which is impossible because of time and resource constraints [2,8,9].

Strategies have been developed in the last 20 years to solve the above problem. Among these strategies, combinatorial testing strategies are the most effective in designing test cases for this problem. These strategies facilitate search and generate a set of tests, thereby forming a complete test suite that covers the required combinations in accordance with the strength or degree of combination. This degree starts from two (i.e., $d = 2$, where d is the degree of combinations).

Considering all combinations in a minimized test suite is a hard computational optimization problem [2,10–12], because searching for the optimal set is an NP-hard problem [2,11–15]. Hence, searching for an optimum set of test cases can be a difficult task, and finding a unified strategy that generates optimum results is challenging. Three approaches, namely, computational algorithms, mathematical construction, and nature-inspired metaheuristic algorithms, can be used to solve this problem efficiently and find a near-optimal solution [16].

Using nature-inspired metaheuristic algorithms can generate more efficient results than other approaches [17,18]. This approach is more flexible than others because it can construct combinatorial sets with different input factors and levels. Hence, its outcome is more applicable because most real-world systems have different input factors and levels. Techniques that have been used to construct combinatorial sets include simulated annealing (SA) [7], tabu search (TS) [19], genetic algorithm (GA) [20], ant colony algorithm (ACA) [20,21], and particle swarm optimization (PSO) [22,23].

SA generates promising results in cases with small parameters and values as well as a small combination degree. However, it could not exceed certain parameters and values, and is unable to obtain results for combination degrees greater than three [20,24]. PSO can compete with other strategies in most cases even when the combination degree exceeds three [25,26]. However, PSO suffers from the effect of parameter tuning on its performance and from problems with local minima. Recent studies have discovered new nature-inspired metaheuristic algorithms that can produce better results than the traditional PSO algorithm for different applications.

Cuckoo Search (CS) [27] is one of the novel nature-inspired algorithms that have been proposed recently to solve complex optimization problems. CS can be used to efficiently solve global optimization problems [28] as well as NP-hard problems that cannot be solved by exact solution methods [29]. The most powerful feature of CS is its use of Lévy flights to update the search space for generating new candidate solutions. This mechanism allows the candidate solutions to be modified by applying many small changes during the iteration of the algorithm. This in turn makes a compromised relationship between exploration and exploitation which enhance the search capability [30]. To this end, recent studies proved that CS is potentially far more efficient than GA and PSO [31]. Such feature have motivated the use of CS to solve different kinds engineering problems such as scheduling problems [32], distribution networks [33], thermodynamics [34], and steel frame design [35].

The current paper presents the design and implementation of a strategy to construct optimized combinatorial sets using CS. Besides the Lévy flights, another advantage of CS over other counterpart nature-inspired algorithms such as PSO and GA, is that it does not have many parameters for tuning. Evidences showed that the generated results were independent of the value of the tuning parameters [27,31].

The rest of the paper is organized as follows: Section 2 presents the mathematical notations, definitions, and theories behind the combinatorial testing. Section 3 illustrates a practical model of the problem using a real-world case study. Section 4 summarizes recent related works and reviews in the existing literature. Section 5 discusses the methodology of the research and implementation. The section reviews CS in detail and discusses the design and implementation of the strategy. In addition, it shows how the combinations are generated and describes in detail the algorithms that are used within the proposed strategy. Section 6 contains the evaluation results on the efficiency, performance, and effectiveness of CS. Section 7 presents threats to validity for the experiments and the case study. Finally, Section 8 concludes the paper.

2. Covering array mathematical preliminaries and notations

One future move toward combinatorial testing involves the use of a sampling strategy derived from a mathematical object called covering array (CA) [36]. In combinatorial testing, CA can be simply demonstrated by a table with rows and columns that contain the designed test cases; each row is a test case, and each column is an input factor for the software-under-test.

This mathematical object originates essentially from another object called orthogonal array (OA) [12]. An orthogonal array $OA_\lambda(N; d, k, v)$ is an $N \times k$ array in which for every $N \times d$ sub-array, each d-tuple occurs exactly λ times, where $\lambda = N/v^d$. In this equation, d is the combination strength; k is the number of factors ($k \geqslant d$), and v is the number of symbols or levels associated with each factor. To consider all combinations, each d-tuple must occur at least once in the final test suite [37]. When each d-tuple occurs exactly one time, then $\lambda = 1$, and it can be excluded from the mathematical notation, i.e., $OA(N; d, k, v)$. As an example, the orthogonal array OA(9; 2, 4, 3) that contains three levels of value (v), with a combination degree (d) of two, and four factors (k) can be generated by nine rows. Fig. 1(a) illustrates the arrangement of this array.

OA (9; 2, 4, 3)

k_1	k_2	k_3	k_4
1	1	1	1
2	2	2	1
3	3	3	1
1	2	3	2
2	3	1	2
3	1	2	2
1	3	2	3
2	1	3	3
3	2	1	3

(a)

CA (9; 2, 4, 3)

k_1	k_2	k_3	k_4
1	3	3	3
3	2	3	1
1	1	2	1
1	2	1	2
3	1	1	3
2	1	3	2
3	3	2	2
2	3	1	1
2	2	2	3

(b)

MCA (9; 2, 4, $3^2 2^2$)

k_1	k_2	k_3	k_4
2	1	1	2
2	2	2	1
3	3	2	2
1	3	1	1
1	1	2	1
1	2	1	2
3	2	1	1
3	1	1	1
2	3	1	2

(c)

Fig. 1. Examples illustrating OA, CA, and MCA.

However, the application of OA is limited by its requirement for uniform factors and levels; thus, this array is suitable for small test suites only [38,39]. To address this limitation, the CA has been introduced to complement OA.

CA is another mathematical notation that is more flexible for representing large test suites with different parameters and values. In general, CA uses the mathematical expression $CA_\lambda(N; d, k, v)$ [1]. A covering array $CA_\lambda(N; d, k, v)$ is an $N \times k$ array over $\{0, \ldots, v-1\}$ such that every $B \in \binom{\{0,\ldots,k-1\}d}{d}$ is λ-covered such that every $N \times d$ sub-array contains all ordered subsets from v values of size d at least λ times [40]. To consider all combinations, d-tuples must occur at least once. As such, we consider the value of $\lambda = 1$, which is often omitted. Hence, the notation becomes $CA(N; d, k, v)$ [41]. We say that the array has size N, combination degree d, k factors, v levels, and index λ. Given d, k, v, and λ, we denote the smallest N for which a $CA_\lambda(N; t, k, v)$ exists as $CAN_\lambda(d, k, g)$. A $CA_\lambda(N; d, k, v)$ with $N = CAN_\lambda(d, k, v)$ is optimal as shown in Eq. (1) [42]. Fig. 1(b) shows a CA with $N = 9$, $k = 4$, $v = 3$, and $d = 2$.

$$CAN(d, k, v) = min\{N :\ni CA(N, d, k, v)\} \quad (1)$$

CA is suitable when the number of levels v is the same for each factor in the array. When factors have different numbers of levels, mixed covering array (MCA) is used. MCA is notated as MCA $(N, d, k, (v_1, v_2, v_3, \ldots, v_k))$. MCA is an $N \times k$ array on v levels and k factors, where the rows of each $N \times d$ sub-array cover all d-tuples of values from the d columns at least once [8]. For additional 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 $(N; d, v^k)$ [14]. Fig. 1(c) shows an MCA with size 9 that has four factors; two of these factors each have three levels, and the other two factors each have two levels, and each of these levels have two values.

3. Real-world problem model

Mozilla Firefox is a practical example that illustrates and models the concepts of combinatorial testing. Mozilla Firefox is a well-known Web browser that has many options and configurations that the user can control without difficulty because of its graphical user interface (GUI). Fig. 2 shows a subset configuration of Mozilla Firefox, when many options of the scheme are combined to create a specific configuration. Configurations exist under various forms that enable them to be controlled in different ways, such as by clicking on the box or checking or unchecking an option. Users can change the configurations by clicking commands while operating Mozilla Firefox. Fig. 2 shows a dialog box that contains six different configurations (i.e., warning when closing multiple tabs and warning when opening these tabs makes the browser operates lowly), with each configuration having two possible values (i.e., check and uncheck).The user can change the configuration based on the requirements.

Testing the program by applying a set of designed test cases may reveal a set of different faults. However, evidence shows that applying the same set of test cases but with different configurations may lead to different faults [43,44], which in turn leads us to consider different configurations for the same software-under-test. In addition, evidence shows that considering the interaction between the configurations (i.e., combination of configurations) will also detect new faults [26].

We need to consider that all the configurations must contain all possible combinations to test the software shown in Fig. 2. Thus, the software has 2^6 configurations, that is, 64 test cases. Xiao Qu called this collection of all possible combinations of configurations *configuration definition layer* (CDL) [43]. Thus, a specific system that contains different configurations must be tested against its CDL, which leads to a configuration-aware testing process. Fig. 3 shows this process.

Ideally, each test case must be run against each configuration of the system. However, for large configurable software systems, considering all configurations is practically impossible because of time and resource constraints. For example, the command language interpreter of the Linux operating system (Bash) has approximately 76×10^{23} possible configurations [44]. Reducing these configurations will dramatically minimize the time and cost of the testing process.

A sampling technique is needed to minimize these configurations systematically. Different sampling techniques are proposed in the literature (see [18,45]). Among those techniques, combinatorial optimization effectively minimizes the number of configurations to be considered based on the combination degree. Combinatorial optimization can also be used to minimize the number of test cases. The final test suite can be represented mathematically by CA notation. The example in Fig. 2 has six factors, each of which has two configurations. Considering combination degree

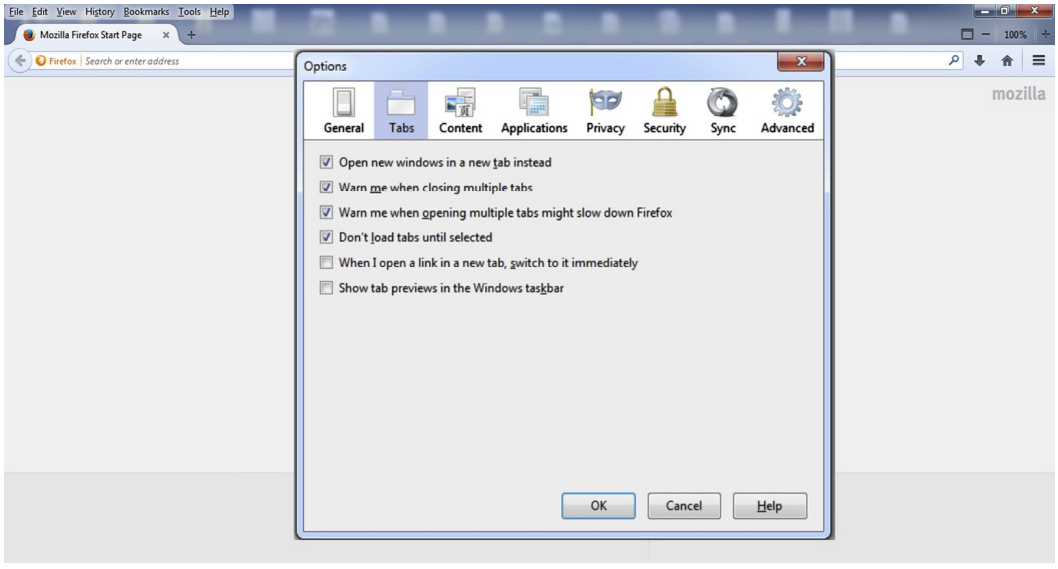

Fig. 2. Subset configuration of Mozilla Firefox.

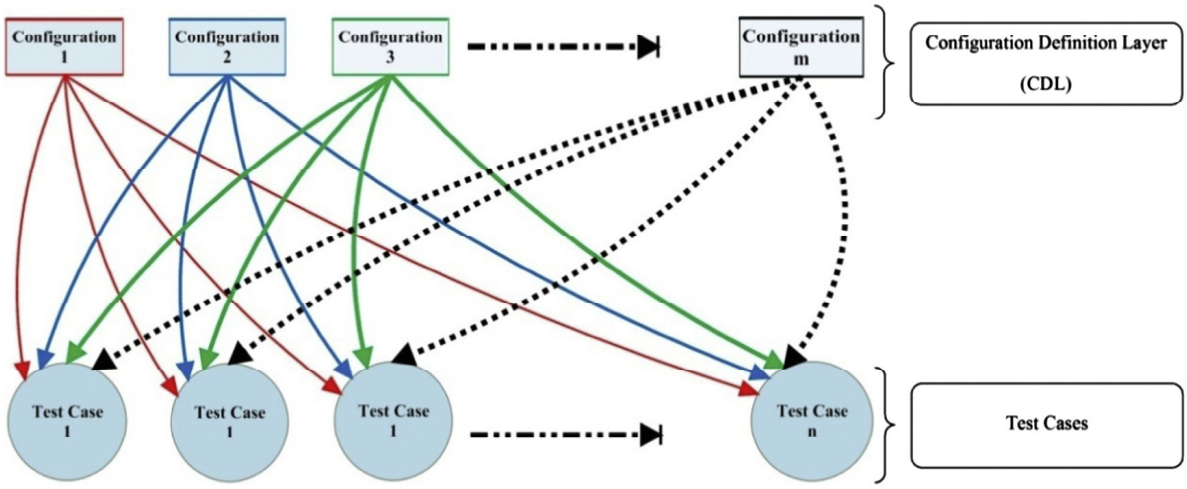

Fig. 3. Relationship between test cases and configurations in configuration-aware testing.

$d = 2$, the configuration set can be minimized to six configurations, that is, CA(6; 2, 6, 2), by covering all the combinations of two configurations. However, CA(13; 3, 6, 2), CA(26; 4, 6, 2), and CA(33; 5, 6, 2) represent configuration sets for combination degrees of 3, 4, and 5, respectively. Thus, instead of selecting all the combinations exhaustively, equivalence sets could lead to improved results with minimized time and cost.

4. Review of the literature and related works

As mentioned previously, generating a CA is an NP-hard problem .Thus, better methods have been sought. From the literature and other evidence, the generation methods have been confined to four main directions, namely, random method, mathematic method, greedy algorithm, and heuristic search algorithm [18].

Random methods are akin to ad hoc generation methods. In most cases, random methods work through a mechanism of random selection of a row of CA and by verifying whether it covers most of the combinations. The method continues to iterate until all the combinations are covered. This method is often used to show the effectiveness of other generation algorithms or to compare its fault detection abilities with other proposed methods [46,47]. Although it obtains better results in some cases, random generation methods usually fail to achieve substantial results [48].

The problem of CA generation has also been solved with extensions of OA construction that involve mathematical functions, regardless of the functions used for construction. Other mathematical methods use a recursive construction approach by building larger CAs from smaller CAs [41,49]. Specifically, different tools use mathematical methods for construction, such as Combinatorial Test Services [50] and TConfig [51]. Although mathematical methods can effectively generate small-sized CAs, they fail to generate CAs for large parameters and values, particularly when the values are unequal among the parameters (i.e., MCA). These drawbacks limit its application for different cases of CA construction. However, the mathematical approach has the advantage of lightweight computation, which means it has a relatively fast generation time. In addition, the mathematical approach can produce optimal CAs for some special cases [40,52,53].

In addition to the aforementioned approaches, greedy algorithms and mathematical search methods solve the problem of CA generation computationally. Greedy algorithms are used to cover many uncovered combinations in each row of the CA. In this study, the CA rows are generated by using either of two methods, namely, one-row-at-a-time or one-parameter-at-a-time [18]. In the one-row-at-a-time method, the CA is constructed row by row. When a row is added, this row should essentially cover all d-tuples as much as possible. The construction process will stop when all d-tuples are covered successfully. The automatic efficient test generator (AETG) [54] is probably the first strategy that adopts this method of generation. The AETG strategy selects greedily one test case among several candidate test cases for each cycle. This algorithm serves as a base for a number of variations that have been developed later for AETG such as mAETG_SAT [7] and mAETG [24]. In addition to AETG, more work has been conducted on developing different algorithms and tools, such as the algorithm used for pairwise generation in the CATS tool [55], the greedy algorithms used in the Pairwise Independent Combinatorial Testing (PICT) tool [56], and the density-based greedy algorithm [57]. Most recently, pseudo-Boolean optimization is used with an AETG-like algorithm to generate efficient test suites [58]. Here, the strategy tries to do not reach maximum coverage of the d-tuples by the test cases. Instead, it tries to reach a balance point for the coverage ratio between [0.8, 0.9].

The one-parameter-at-a-time method attempts to construct rows of the generated CA by adding one parameter to each row each time and verifying the coverage of d-tuples periodically. Based on the coverage, parameters are added to the rows horizontally and vertically using heuristics until the CA is completed. The in-parameter-order (IPO) algorithm [10] was the pioneering implementation of this method. This strategy was further developed to produce variations of the IPO algorithm, such as IPOG [59], IPOG-D [14], IPOG-F [60], and IPO-s [61].

Heuristic search and artificial intelligence (AI)-based techniques have been applied effectively for CA construction. In general, these techniques start with a random set of solutions. Then, a transformation mechanism is applied to this set such that it is transferred to a new set in which its solutions are more efficient for d-tuple coverage. For each iteration, the transformation equations essentially create a more efficient set. Despite detailed variations in the heuristic search techniques, they essentially differ in transformation functions and mechanisms. Here, techniques, such as SA [7], TS [19], GA [20], ACA [20,21], and PSO [22,23], were used effectively for CA construction.

From a practical point of view, most of the time, the input factors of the real world applications suffer from the intertwined dependencies among each other which can potentially lead to problem in executing the test cases and may lead to failure due to improper execution [62]. Here, some of the parameters combinations are considered as impossible combination. Hence, they are considered as constraints in which they must be part of the final test suite. To this end, some of the recent strategies and tools

start to support this issue such as mAETG, mAETG_SAT, IPOG, IPOG-D, SA, PICT, and TVG. However, strategies like GA, ACA, and PSO) generally do not show any evidence to support constraints. Recently, Garvin, B., et al. improve the SA algorithm to support constrained interaction testing [63]. To add the support for constraints, it is required to remove those combinations from the d-tuples list and add them directly to the final test suite.

Evidence showed that the computational methods (i.e., greedy and heuristic search algorithms) generate better results in terms of size. However, the computational methods may require more computational time than mathematical and random methods. In addition, the computational approach is more flexible than the other approaches as it can construct CAs with different parameters and values. Thus, the outcome of computational methods is more applicable because most real-world systems have different parameters and values rather than equal parameter values. Nonetheless, mathematical methods are useful for generating the optimal construction of CA in cases with few parameters and values, and a low degree of combination. As a result, computational methods are more applicable and more realistic, although they may not consistently produce the optimal CA.

As mentioned previously, different metaheuristic and AI-based strategies are proposed in the literature. Given an NP-hard problem, deriving a strategy that can generate optimal test cases for all parameters and values is practically impossible. To this end, researchers have attempted to construct better CAs in terms of size for most cases and to overcome the drawbacks of each method. In the case of small parameters and values, as well as a small combination degree d, SA usually generates promising results most of the time. However, SA is less effective when $d > 3$. GA, ACA, and TS have also been applied in previous studies for generation [19,20]. By contrast, PSO can compete with other strategies when $d > 3$ [25,26]. However, PSO suffers from problems, such as parameter tuning, sticking in the local minima, and premature convergence of swarm problems which affect its optimization capability.

Particularly, these strategies suffer from heavy computation and inaccurate results for combinatorial test suite generation. For example, GA suffers from the crossover and mutation processes, which lead to heavy computation, ACA suffers from different problems when the number of ants increases, and TS suffers from the update mechanism of the tabu list sets. In addition, these strategies often impose a trade-off between reliability and speed of computation [34]. Today, no metaheuristic strategy can generate optimized results for all configurations, thereby implying that the investigation of new and efficient strategies with the help of metaheuristics is still an active research topic.

Cuckoo Search (CS) has recently been found to be effective in solving engineering and optimization applications, with promising results. The convergence characteristics and results of CS are better than those of other metaheuristic optimization methods [28,64]. In the literature, no studies have applied this promising method to generate combinatorial test suites. Thus, in this paper, we attempt to modify and apply the relative strengths of CS to this important part of software testing.

5. Cuckoo Search for combinatorial testing

Generating effective test cases and configurations is the most challenging task. As mentioned previously, testing the application exhaustively (i.e., test every possible event) is impossible most of the time because of time and resource constrains. Thus, an optimization strategy is needed to optimize and generate an optimized test suite that has the effectiveness of exhaustive testing. In this study, we use CS to search for test cases that cover all possible combinations at least once.

In this section, we provide the necessary details for the developed strategy. Section 5.1 presents the necessary background and illustrates the essential details of CS and its mechanism. Section 5.2 presents the details of the "all-combination-list generation" algorithm. Then, Section 5.3 presents the CS used for combinatorial testing and its optimization process and implementation.

5.1. Cuckoo Search (CS)

CS is a new metaheuristic search algorithm that was developed by Yang and Deb [27]. CS is inspired by the behavior of a fascinating bird called the cuckoo. The aggressive reproduction strategy of this bird inspired the researchers to study and investigate the opportunity to use its behavior within an optimization mechanism. Cuckoos lay their eggs in communal nests, although they may remove the eggs of another bird to increase the hatching probability of their own eggs. If the host bird discovers the eggs of the cuckoo, then it may throw the eggs away from the nest or may completely abandon the nest. The physiology and behavior of the cuckoo have the capability to mimic the appearance of the egg of the host.

Algorithm 1: Cuckoo Search

```
Initialize a population of n host nests x_i , i = 1, 2, ... , n
for all x_i do
    Calculate fitness F_i = f(x_i)
end
while (Number of iterations < Max Number of iterations)
    or (Stopping criteria satisfied) do
    Generate a cuckoo egg (x_j) by taking a Lévy flight from random nest
    F_j = f(x_j)
    Choose a random nest i
    if F_i > F_j then
        x_i ← x_j
        F_i ← F_j
    end
    Abandon a fraction pa of the worst nests
    Build new nests at new locations via Lévy flights to replace nests lost
    Evaluate fitness of new nests and rank all solutions
end
```

Fig. 4. Pseudo code of Cuckoo Search [27].

The rules of the CS are as follows: (1) Each cuckoo selects a nest randomly to lay one egg in it, in which the egg represents a solution in a set of solutions. (2) Part of the nest contains the best solutions (eggs) that will survive to the next generation. (3) The probability of the host bird finding the alien egg in a fixed number of nests is $p_a] \in [0,1]$ [65]. If the host bird discovers the alien egg with this probability, then the bird will either discard the egg or abandon the nest to build a new one. Thus, we assumed that a part of p_a with n nest is replaced by new nests. Fig. 4 shows the pseudocode and steps of the algorithm [66].

Lévy flight is used in the cuckoo algorithm to conduct local and global searches [67]. Here, Lévy flight serves as an update mechanism to update and modify the initial random search space. This update mechanism allows the algorithm to generate new candidate solutions by applying small changes during the iteration which behave like a step toward the best solution [30]. The rule of Lévy flight is used successfully in stochastic simulations of different applications, such as biology and physics. Lévy flight is a random path of walking that takes a sequence of jumps, which are selected from a probability function. A step can be represented by the following equation for the solution $x^{(t+1)}$ of cuckoo i:

$$x_i^{t+1} = x_i^{(t)} + \alpha \oplus \text{Lévy } (\lambda) \tag{2}$$

where α the size of each step in which $\alpha > 0$ and depends on the optimization problem scale. The product $\oplus$ is the entrywise multiplication, and Lévy (λ) is the Lévy distribution. The algorithm continues to move the eggs to another position if the objective function found better positions.

Another advantage of CS over other counterpart stochastic optimization algorithms, such as PSO and GA, is that it does not have many parameters for tuning. The only parameter for tuning is p_a. Yang and Deb [27,31] obtained evidence from the literature and showed that the generated results were independent of the value of this parameter and can be fit to a proposed value $p_a = 0.25$.

5.2. The d-tuples list generation algorithm

Generating the d-tuples list is essential to calculate the fitness function $F_i = f(x_i)$. The d-tuples list contains all possibilities of combinations between input factors k. As an example, we consider a system with three input factors $(k_1 k_2 k_3)$, each factor having three levels $(v_1\, v_2\, v_3)$. For exhaustive testing, when the combination degree $d = 3$ (i.e., $d = k$), $(3 \times 3 \times 3)$ combinations result in 9 combinations. However, as mentioned previously, exhaustive testing is impossible. Thus, lower combination degrees are considered to minimize the test cases. For example, when $d = 2$, the combinations are $(k_1 k_2)$, $(k_1 k_3)$, and $(k_2 k_3)$. In turn, these combinations are converted to the all-combination-list, which contains $(3 \times 3) + (3 \times 3) + (3 \times 3) = 27$ combinations with $d = 2$. Then, this list will be covered row by row during the optimization process.

Generation this list is difficult because of its tightness with the combination degree. Thus, the generation of the list starts by considering the number of factors and then calculating the binary equivalence numbers of $(2^k - 1)$. This algorithm is implemented in the "Generate Binary Digits" function, as shown in Fig. 5.

The algorithm starts by inputting binary digits from 0 to $(2^k - 1)$ in a list. For example, when $k = 3$, then the list contains (000) to (111). A filtering mechanism is combined with the algorithm to filter the number of (1's) in each number from the list depending on the combination degree. For example, when $d = 2$, then the binary numbers after filtering are $[(011), (101), (110)]$, which are equivalent to $[(k_2 k_3), (k_1 k_3), (k_1 k_2)]$, which, in turn, serves as a master algorithm for generating combinations of input factors for all degrees. The progress and output of this algorithm can be noted clearly in the output screen of the strategy shown in Fig. 6.

When the combinations of factors are identified, the values of the corresponding factor are matched. This algorithm is implemented in the "Generate-Combinations" function, as shown in Figs. 5 and 6. When a factor is missed in the combination (i.e., its corresponding binary value is 0), its corresponding value will be "don't care" because we do not need its value for that specific combination (in this study, -1 is used as an indication only). Upon completion of this algorithm, all the combinations are stored in a list to be used for calculating the fitness function of the CS. The output list of this algorithm can be noted clearly in the output screen of the strategy shown in Fig. 6.

An algorithm is used to assess the search process for the combinations efficiently. In this study, the rows in the d-tuples list are stored in groups. Each group is assigned an index number that indicates its position in the list. The groups are selected based on the combination of factors. For example, in the aforementioned sample, the combination $(k_2 k_3)$ is stored in the index from 0 to 8 because it has nine rows of combinations. Thus, the next group is stored in the index from 9 to 17.

5.3. Optimization process with Cuckoo Search

When the d-tuples list is generated, then CS starts. In this study, the CS algorithm is modified to solve the current problem. The

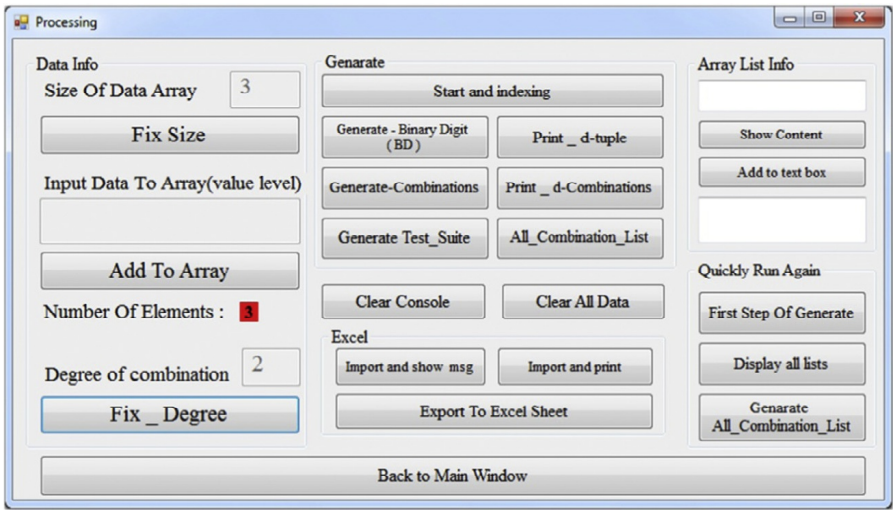

Fig. 5. Main window of the implemented strategy.

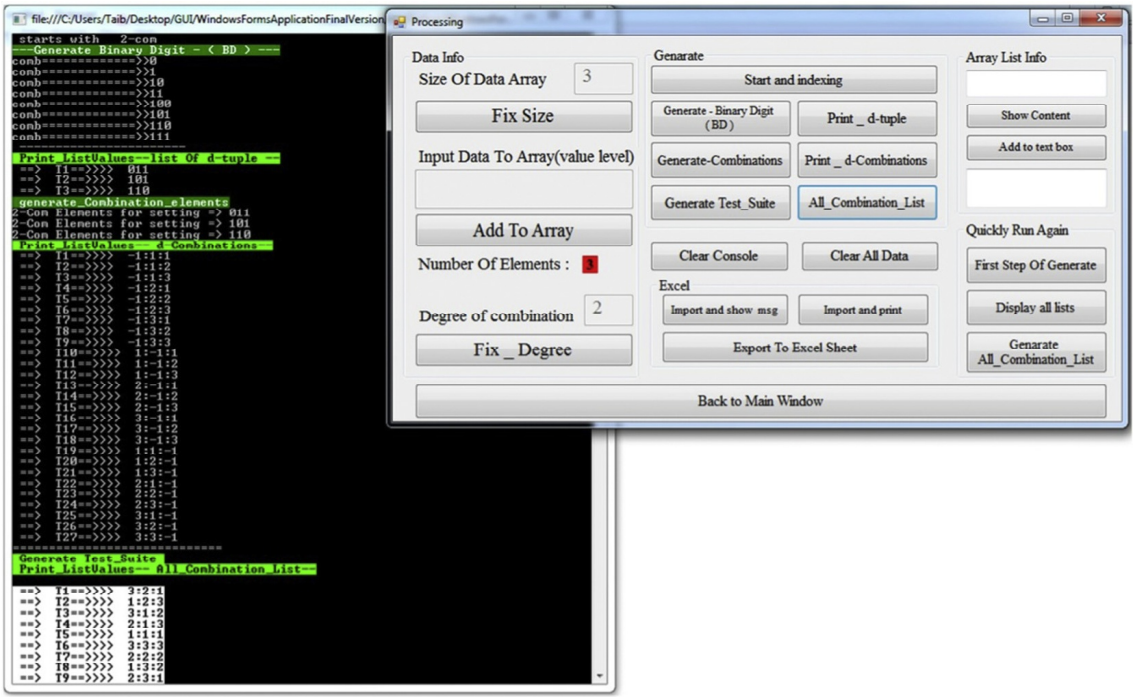

Fig. 6. Strategy in progress when each algorithm is executed and the final optimized set is generated.

fitness function is used to derive the better solution among a set of solutions. In this study, a row with higher fitness weight is defined as a row that can cover a higher number of rows in the *d*-tuples list. Fig. 7 shows the pseudocode of combinatorial test suite generation in which the CS is modified for this purpose.

As shown in Fig. 7, the strategy starts by considering the input configuration. Then, the *d*-tuples list is generated. CS starts by initializing a random population that contains a number of nests. Given that the number of levels for each input factor is a discrete number, the initialized population is discrete, not an open interval. Thus, the population is initialized with a fixed interval between 0 and v_i. In this study, a system has different factors in which a test case is a composite of more than two factors that form a row in the final test suite. As a result of such an arrangement, each test case is treated as a vector x_i that has dimensions equal to the number of input factors of the system. In addition, the levels for each input factor are basically an integer value. As a result, each dimension in the vector-initialized population must be an integer value.

Although the initial population is initialized in a discrete interval, the algorithm can produce out-of-the-bound levels for the input factors. Thus, the vector must be restricted with lower and upper bounds. The rationale behind this restriction is that the cuckoo lays its eggs in the nests that are recognized by its eyes.

When the CS iterates, it uses Lévy flight to walk toward the optimum solution. Lévy flight is a walk that uses random steps in which the length of each step is determined by Lévy distribution. The generation of random steps in Lévy flight consists of two steps [68], namely, the generation of steps and the choice of random direction. The generation of direction normally follows a uniform distribution. However, in the literature, the generation of steps follows a few methods. In this study, we follow the Mantegna algorithm, which is the most efficient and effective step generation method. Within this algorithm, a step length s can be defined as follows:

$$s = \frac{u}{|v|^{1/\beta}} \tag{3}$$

where u and v are derived from the normal distribution in which

$$u \sim N(0, \sigma_u^2) \quad v \sim N(0, \sigma_u^2) \tag{4}$$

$$\sigma_u = \left\{ \frac{\Gamma(1+\beta)\sin\left(\frac{\pi\beta}{2}\right)}{\Gamma\left[\frac{1+\beta}{2}\right]\beta 2^{\frac{\beta-1}{2}}} \right\}^{1/\beta}, \quad \sigma_v = 1 \tag{5}$$

Based on the aforementioned design constraints, the complete strategy steps, including the CS, are summarized in Fig. 7.

As mentioned previously, the strategy starts by considering the input configuration. Normally, the input is a composite input with the factors, levels, and desired combination degree *d*. The combination degree $d > 1$ and is less than the number of input factors. Using the *d*-tuples generation algorithm described previously, the *d*-tuples list is generated (Step 1). Then, the strategy uses the CS, which starts by initializing a population with m nests, with each nest consisting of dimensional vectors equal to the number of factors that have a number of levels (Step 2). From a practical point of view, each nest contains a candidate test case for the final test suite. Then, the CS starts to assess each nest by evaluating coverage capability of the *d*-tuples (Steps 3–5). This mechanism is used to assess the fitness function of the CS. The fitness function $f(x_i)$ of the test case x_i in this strategy is defined as:

$$f(x_i) = \sum_{i \in newd-tuple(x_i)}^{i} d_i \tag{6}$$

where *d-tuples*(x_i) indicates new tuples that are not covered by the previous generated tests but covered by the test x_i. d_i denotes the strength of the interaction *i*. For example, when a nest can cover four *d*-tuples, then its weight of coverage is 4. The strategy uses a special mechanism described previously (Section 5.2) to determine the number of covered tuples and to verify the weight. Based on the results of coverage for all nests, the strategy sorts the nests again in the search space based on the highest coverage (Step 10). The lowest coverage in the search space will be abandoned. For the abandoned nests, a Lévy flight is conducted to verify the availability of better coverage (Step 11). If better coverage is obtained for a specific nest, then the nest is replaced by the current nest content (Steps 12–15). This process serves just like global search in other optimization algorithms. Then, for all of the top nests after sorting, a Lévy flight is conducted to search for the local best nests (Steps 17–19). If better coverage is obtained after the Lévy flight for a

Algorithm 2: Combinatorial test suite generation

Input: Input-factors k and levels v
Output: A test case

```
1  Let d-tuples list be a set of all combinations' list that must be covered
2  Initialize a population of m host nests x_i, i = 1, 2, ..., m
3  for all x_i do
4  |  Calculate the coverage of combinations and return the weight
5  end
6  Iteration number Iter ← 1
7  while (Number of iterations <Max Number of iterations)
8      or (d-tuples list is not empty) do
9  |  Iter ← Iter + 1
10 |  Sort the nest by the weight of combination's coverage
11 |  for all nests to be abandoned do
12 |  |  Current position x_i
13 |  |  Perform Lévy flight from x_i to generate new egg x_j
14 |  |  x_i ← x_j
15 |  |  F_i ← f(x_i)
16 |  end
17 |  for all of the top nests do
18 |  |  Current position x_i
19 |  |  perform Lévy flight from x_i to generate new eggs x_k
20 |  |  F_k ← f(x_k)
21 |  |  if F_k > F_i then
22 |  |  |  x_i ← x_k
23 |  |  |  F_i ← F_k
24 |  |  end
25 |  end
26 end
27 Add the first nest to the final test suite
28 Remove all the related combinations from the d-tuples list
```

Fig. 7. Pseudocode of combinatorial test case generation with CS.

specific nest, then the nest is replaced with the one that have better coverage (Steps 20–24). These steps (Steps 9–26) in the CS will update the search space for each iteration.

Two stopping criteria are defined for the CS. First, if the nest reaches the maximum coverage, then the loop will stop and the algorithm will add this test case to the final test suite and remove its related tuples in the *n*-tuples list. Second, if the *d*-tuples list is empty, then no combinations are covered. If the iteration reaches the final iteration, then the algorithm will select the best coverage nest to be added to the final test suite (Step 27) and remove the related tuples in the *n*-tuples list (Step 28). Fig. 8 shows a graphical representation of the strategy to summarize the aforementioned steps for better understanding. The sequence of running is show in red[1] circle in the figure.

The constructed final test suite can be noted clearly in Fig. 6. This mechanism will continue as far as *n*-tuples remain in the list. Fig. 9 shows a running example to illustrate how the tuples are covered and removed and how the final test suite is constructed.

6. Evaluation results and discussion

The evaluation phase for the proposed strategy is divided into the following three main sections: (1) evaluation of the generation efficiency, (2) evaluation of the generation performance, and (3) evaluation of the effectiveness of the generated test suite. Based on the literature [16,26,69], efficiency is evaluated based on the size of the generated test suite, whereas performance is evaluated based on the time taken by the strategy to generate a specific test suite. For these two evaluation phases, the strategy is compared with other available strategies.

Some strategies are available publicly as tools to be downloaded and installed. Other strategies are unavailable publicly, yet their evaluations are published for certain cases. We consider the performance evaluation for strategies that are available for implementation within the same evaluation environment. By contrast, for unavailable strategies, we consider the efficiency evaluation only. The rationale behind this option is that the efficiency criterion is not affected by the research environment as the size of the CA is not affected by computer speed. However, installing all the tools in the same environment is essential to ensure a fair comparison of performance as the construction time is affected by the specifications of the computer.

The effectiveness of the generated test suite is evaluated by adopting a case study on a reliable artifact program to prove the applicability and correctness of the strategy for a real-world software testing problem. Given that the generated test suite did not consider the internal structure of the artifact program, the testing process represents a functional testing process that considers the program configuration.

The experimental environment consists of a desktop PC with Windows 7, 64-bit, 2.5 GHz, Intel Core i5 CPU, and 6 GB of RAM. The algorithms are coded and implemented in C#.

6.1. Efficiency evaluation

The efficiency of the combinatorial test suite construction is measured by the size of the test suite generated by the strategy. For strategies that depend on metaheuristic algorithms, a degree of randomness is observed, especially when the strategy starts

[1] For interpretation of color in Figs. 8 and 11, the reader is referred to the web version of this article.