This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

Constructing Cost-Aware Functional Test-Suites Using Nested Differential Evolution Algorithm

Yuexing Wang, Min Zhou, Xiaoyu Song, Ming Gu, Jiaguang Sun

***Abstract*—Combinatorial testing can test software that has various configurations for multiple parameters efficiently. This method is based on a set of test cases that guarantee a certain level of interaction among parameters. Mixed covering array can be used to represent a test-suite. Each row of the array corresponds to a test case.**

In general, a smaller size of mixed covering array does not necessarily imply less testing time. There are certain combinations of parameter values which would take much longer time than other cases. Based on this observation, it is more valuable to construct mixed covering arrays that are better in terms of testing effort characterization other than size. We present a method to find cost-aware mixed covering arrays. The method contains two steps. First, simulated annealing algorithm is used to get a mixed covering array with a small size. Then we propose a novel nested differential evolution algorithm to improve the solution with its testing effort. The experimental results indicate that our method succeeds in constructing cost-aware mixed covering arrays for real-world applications. The testing effort is significantly reduced compared with representative state-of-the-art algorithms.

***Index Terms*—Combinatorial testing, mixed covering arrays, simulated annealing, nested differential evolution.**

I. Introduction

TO ensure the reliability of software, some researchers use qualified code generation tools from high-level model to ensure the correctness at the early design stage[1][2], and some others use system test techniques. For the latter, there are many methods to test software that has various configurations. Traditional methods test all combinations of parameter values. They are very costly because the number of configurations simply grows exponentially with respect to the number of parameters.

Combinatorial Testing (CT) is proposed to reduce the scale of test cases. CT analyzes interactions among software's parameters using a very small number of test cases. The idea is that a fault is often caused by interactions among a small number (say t) of parameters [3]. Thus it is important to test all combinations of every t parameters. Kuhn $et\ al.$ [4][5] studied the faults in several software projects and found that all known faults in these projects can be caused by testing interactions among t $(2 \leq t \leq 6)$ parameters [4][5]. CT can be applied to test software efficiently because it can reduce the scale of test cases.

Y. Wang, M. Zhou(corresponding author), M. Gu, and J. Sun are with the School of Software, Tsinghua University, Key Laboratory for Information System Security, Ministry of Education and Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China. E-mail: wang-yx15@mails.tsinghua.edu.cn, zhoumin03@gmail.com

X. Song is with the Electrical and Computer Engineering Dept, Portland State University, Portland, OR 97201, USA. E-mail: songx@pdx.edu

TABLE I: Hypothetical e-commerce system example

	Browser	WebServer	Database	Payment
0	Firefox	Apache	MySQL	Visa
1	Chromium	IIS	MaxDB	MasterCard
2	IE	WebSphere		

Combinatorial testing uses Covering Arrays (CA) and Mixed Covering Arrays (MCA) to represent test-suites. A CA is a matrix of $N \times k$ that is denoted by $CA(N; t, k, v)$. N is the number of test cases (number of matrix rows). k is the number of software parameters (number of matrix columns) and each parameter has v possible values. t is the $strength$ or $degree$ of interactions. Each valid CA should have the property such that every $N \times t$ subarray contains all possible combinations of the corresponding t parameters. Researchers have shown that finding size optimal CAs is NP-complete [6][7][8].

MCA is the same as CA except that the number of each parameter's possible values can be different from each other. MCAs are more appropriate to represent test-suites of software than CAs. A $strength$-t MCA is denoted by $MCA(N; t, k, v_1 v_2 ... v_k)$, N, t, k are the same as CA. The corresponding set of values that the k parameters can get from is $(S_1, ..., S_i, ..., S_k)$, which has $(v_1, ..., v_i, ..., v_k)$ elements. MCAs which have smaller sizes are called $size$-$aware$ MCAs and which require less testing effort are called $cost$-$aware$ MCAs. MCA has the following two properties [3]:

1) The values appear in column i are contained in the set S_i with $v_i = |S_i|$.

2) Every $N \times t$ subarray contains all the possible combinations of the corresponding t parameters at least once. One such combination is called a t-$tuple$.

To demonstrate the use of MCA in software testing, we introduce a hypothetical e-commerce system [3] in Table I. The system has four parameters, two of which have three possible values and the other two have two possible values. In order to test the system using traditional method, 3*3*2*2=36 test cases are required. However, if we use CT (the strength of interaction is 2) to test the system and use MCA to represent the corresponding test cases, as we can see in Fig. 1, we only need 9 test cases. Fig. 1 is an MCA corresponding to $MCA(9; 2, 4, 3^2 2^2)$ and is translated into test suite in Table II. The rule to translate MCA's rows into test cases is easy. For each parameter, we use its value's row number in Table I to represent the value. So each row of MCA can be translated into one test case.

The time required for a test case to be tested is called the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

testing time of this test case. The time required for all the test cases represented by an MCA to be tested is called the testing time of this MCA.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 \\ 2 & 2 & 0 & 0 \\ 2 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Fig. 1: $MCA(9; 2, 4, 3^2 2^2)$

Until now, many methods have been applied to find MCAs. To the best of our knowledge, all existing methods try to find size-aware MCAs because these MCAs have a smaller size. Thus the testing process would cost less time. Researchers try to decrease the number of test cases to reduce testing time.

However, smaller size does not always mean less testing time. Finding MCAs with small sizes is reasonable only if all the test cases cost the same length of time to be tested. It is possible that some rows in an MCA (representing test cases) take a lot of time to be tested while the other rows cost less. For example, 'CPAchecker' is a configurable tool for software verification. We found that under some configurations, CPAchecker needs more than 800s to be tested using just one file. However, there are also some configurations that can replace the configurations above and only need less than 10s to be tested. Thus we can replace these rows by more other rows while the new MCA's total testing time decreases. The new MCA might have more rows but it needs less testing time. Based on this observation, it is more valuable to construct such new cost-aware MCAs than normal MCAs.

We proposed a new method to construct cost-aware MCAs. The method contains two steps. Firstly, we use an existing simulated annealing (SA) algorithm to construct MCAs with small sizes as initial solutions. Then a novel nested differential evolution (NDE) algorithm is presented to find cost-aware MCAs based on initial solutions.

Contributions of this paper are listed below:

1) We propose a new metric to find the cost-aware MCA instead of size-aware one, which is more accurate for controlling testing efforts.

TABLE II: The software's test suite shown in Fig. 1

	Browser	WebServer	Database	Payment
1	Chromium	IIS	MySQL	Visa
2	Firefox	IIS	MaxDB	MasterCard
3	Chromium	WebSphere	MaxDB	MasterCard
4	IE	Apache	MySQL	MasterCard
5	Firefox	WebSphere	MySQL	Visa
6	IE	WebSphere	MySQL	Visa
7	IE	IIS	MaxDB	Visa
8	Firefox	Apache	MaxDB	MasterCard
9	Chromium	Apache	MaxDB	Visa

2) We propose a novel nested differential evolution algorithm to find cost-aware MCAs efficiently.

3) We apply our method on benchmarks and real-world applications. The results are quite promising.

The rest of this paper is organized as follows, Section 2 gives related work. SA algorithm as well as methods to get testing time functions used in our method are shown in Section 3. Section 4 describes the framework of our NDE algorithm and the improvements. Section 5 shows the experimental results and Section 6 gives the conclusions.

II. Related work

Researchers have paid many efforts to ensure the reliability of software, where some researchers use qualified code generation tools and some others use system test techniques. For the former, there are lots of tools for generating reliable hardware and software codes from formal verified model[9][10] accompanied by static analysis methods using SMT solvers with some optimizations[11][12] or dynamic analysis methods[13][14] for further reliability enhancement. For the latter, researchers have proposed many efficient ways to construct MCAs, such as algebraic methods, greedy methods, and metaheuristic methods.

Algebraic methods [15][16] are proposed to construct strength-2 MCAs. Charles et al. [15] proposed a cut-and-paste method to construct strength-2 CAs. CAs found by their method admit different numbers of symbols in different columns. Williams et al. [16] presented a method based on orthogonal latin squares to generate test configurations that achieve pair-wise parameter coverage.

Some tools such as AETG, TCG, ACTS and DDA are introduced in literature [17][18][19][6]. These tools use greedy algorithms to construct MCAs. Aldiwan et al. presented TCG [18]. It can automatically generate a set of test cases that guarantee a selected interaction degree among parameters. Yu Lei and his partners [19] present a tool aims to generate not only strength-2 CAs but also strength-t ($2 \leq t \leq 6$) CAs. The authors proposed a strategy, called In-Parameter-Order, from pairwise testing to t-way testing.

Metaheuristic algorithms such as simulated annealing (SA) algorithm [3], tabu search algorithm[20] and differential evolution algorithm[21], can construct size optimal or near size optimal MCAs. Avila-George et al. proposed an efficient SA algorithm to construct MCAs with small sizes [3]. Tabu search (TS) algorithm was also proposed to construct MCAs [20]. The novelty in the algorithm was a mixture of three carefully designed neighborhood functions. A new benchmark was also presented. Differential evolution algorithm was also used in the problem of finding size-aware MCAs[21].

Though there are many methods to construct MCAs, most of them have some limitations. For algebraic methods[15][16], the weak point is that they are specially designed for strength-2 MCAs so they cannot be used to construct MCAs with larger strength. Besides, there are some preconditions we need to consider before we use these methods. Most time greedy methods [17][18][19][6] cannot generate MCAs with small sizes. But in real-world software testing process, a few test

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

cases might need a lot of time to be tested and thus the results of greedy methods are not good. As to metaheuristic methods [3][22][7][20], they can construct MCAs optimal or near optimal in sizes. However, we believe that considering the total testing time of an MCA is better than considering its size. Because reduce the size of an MCA aims to reduce the total testing time of it. So total testing time is more worthy of consideration. Algebraic methods and greedy methods also suffer from the same problem.

III. Simulated Annealing and Time function

In this section, we give the first step of our algorithm. First SA which is used to find MCAs with small sizes is presented. Then we propose some methods to get the testing time function of the tested software. Both the MCA found by SA and the testing time function are used in the second step.

A. Find a minimal MCA

Our method has two steps and the first step is presented in detail in this section. In the first step, we want to find $MCA(N, t, k, v_1 v_2 ... v_k)$ with a smaller size. To this situation, an existing SA algorithm is used to construct MCAs as small as possible. We choose SA because it is quite competitive in this problem [3]. The algorithm's pseudocode is shown in Algorithm 1. Let $(p_1, ..., p_i, ..., p_k)$ be the k parameters of a software and the corresponding set of values that the parameters can get from is $(S_1, ..., S_i, ..., S_k)$, which has $(v_1, ..., v_i, ..., v_k)$ possible values. The interaction degree among parameters is t.

We use N to represent the solution's size and the initial solution's size is set to $N = max(\prod_{i=1}^{t} v_{c_i}), \forall c_1, c_2, ..., c_t | 1 \leq c_1 < c_2 < ... < c_t \leq k$. The algorithm starts by generating an initial solution using Hamming distance. Let x, y be two vectors of size k, then the Hamming distance $d(x, y)$ between the two vectors is defined as the number of elements in which they differ. Select one row r_i from the MCA, its Hamming distance is defined by

$$h(r_i) = \sum_{s=0}^{i-1} d(r_i, r_s) \quad (1)$$

In order to generate an initial solution, the following steps are performed:

1) Generate one row at random as the first row.

2) Generate p rows $c_1, c_2, ..., c_p$ as candidate rows.

3) Choosing one row which has the largest Hamming distance calculated by (1) from the p rows.

4) Repeat step2 to step4 N-1 times. The generated N rows formed the initial solution M.

1) Initial solution:

2) Evaluation function: Let M be a solution and $E(M)$ be the evaluation function. $E(M)$ returns the number of t-tuples which are not covered by M. The total number of all the possible t-tuples is given by (2):

$$\sum_{\forall p_1, p_2, ..., p_t | 1 \leq p_1 < p_2 < ... < p_t \leq k} \prod_{i=1}^{t} v_{p_i} \quad (2)$$

Algorithm 1 Simulated_Annealing

```
1: procedure SA
2:     Initialize(M, T, L)
3:     while !stopcriterion do
4:         for l ← 1 to L do
5:             M' ← GeneateNeighbor(M)
6:             if E(M') < E(M) then
7:                 M ← M'
8:             else
9:                 if P > random(0, 1) then
10:                    M ← M'
11:                end if
12:            end if
13:            Calculate(T, φ)
14:        end for
15:    end while
16: end procedure
```

If $E(M') < E(M)$, then M is updated to M'. Otherwise there is also an opportunity that M is updated to M'.

3) Neighborhood functions: There are two types of neighborhood functions: $N_1(M)$ and $N_2(M)$. In each iteration, the probability of using $N_1(M)$ is P_1 and the probability of using $N_2(M)$ is P_2. Thus P_1+P_2 = 1.

The function $N_1(M)$ randomly selects two integers i and j. Then try every other possible value of $M_{i,j}$ and select the one which minimizes $E(M')$ to get a new solution M'.

The function $N_2(M)$ randomly choose t columns and verifies whether all the t-tuples in these columns are covered by M. If the answer is yes, then the next t columns are considered. If the answer is no, it means there exists a t-tuple in these columns that is not covered by M. Suppose the rows in M are $r_0, r_1, ..., r_{N-1}$, then we replace $r_0's$ corresponding t columns with the missing t-tuple and we get a new row r_{0new} and a new solution M_{0new}. Then $r_1, r_2, ..., r_{N-1}$ are handled like that. We select the new row which minimizes $E(M_{*new})$ to get a new solution M'.

4) Cooling schedule: The largest number of permitted iterations is determined by cooling schedule. Cooling schedule uses parameters like an initial temperature T_0, a final temperature T_f and a cooling factor α. There is a relation between T_k and T_{k-1}, T_k = α*T_{k-1}. If T_k is smaller than T_f then the algorithm ends. In each temperature the maximum number of neighborhood solutions that are generated to update M is L. L is an important factor which is critical to the SA algorithm's performance and is determined by N, k and $V(V = max(\prod_{i=1}^{t} v_{c_i}), \forall c_1, c_2, ..., c_t | 1 \leq c_1 < c_2 < ... < c_t \leq k)$.

5) Termination condition: The algorithm terminates when we find an MCA or T_i is smaller than T_f or the solution does not change in ϕ iterations. If it is the latter two situations, then the solution's size N is increased and the algorithm continues until an MCA is found.

More details can be viewed from [3].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

B. Testing time function

To compute the testing time of the whole MCA, We need to know the testing time of each test case. Thus we need a function that measures testing time consumption under different combination of arguments. An approximation of testing time function can be obtained by predicting from a small number of samples.[23][24]. Suppose the real function is f and the predicted function is f'. The set of sample test cases is S. $size(S)$ represents the number of test cases in S. The functions provided to our method should satisfy the following properties:

1) $\delta(f, f') = \sqrt{\frac{\sum_{s \in S}^{S}(f(s) - f'(s)^2)}{size(S)}}$

2) $\delta(f, f') \leq C\varepsilon$

C and ε can be specified different values based on the required precision. The less $C\varepsilon$ is, the more explicit f' will be. Our algorithm can get accurate results if the provided f' is explicit enough.

There are many methods to predict performance of systems. For example, Blais $et\ al.$ [23] proposed an algorithm based on Fourier transform that is able to make predictions of any configurable software system with theoretical guarantees of accuracy and confidence level. Guo, Czarnecki $et\ al.$ [24] presented a variability-aware approach to performance prediction via statistical learning. Using these methods we can get accurate testing time functions. Some easy ways are carried out as alternatives to get testing time functions:

1) If we have the source code of the tested program and the program is easy, we can easily get the relations between testing time and test cases.

2) In most situations, We can not get the function mentioned above easily, but a function can be fitted through a small amount of existing data. For example, we can use linear regression method to get testing time functions based on existing sample test cases. We use this method in our paper.

3) We can use MCAs in regression testing phase. Since We have got the time data earlier, the function is no longer needed.

In our experiments, C and ε are defined as follows:

$C = \max(f(s) - f(s'))$, $s, s' \in S$; $\varepsilon = 0.15$

Note that s and s' are different configurations of a system and the size of S is 100. For CPAchecker C is 800-10=790, for pigz C is 180-7=173 and for GNU Make C is 70-10=60.

IV. Nested Differential Evolution

DE is considered the most suitable algorithm to find cost-aware MCAs based on initial solutions. We should reconstruct initial solutions by substituting a single test instance with multiple test instances assuring that the total testing time decreases and the same unique tuples been covered. It's an optimization problem and the search space can be very large, thus metaheuristic algorithms are preferred. Some researchers gave comparisons between multiple algorithms like DE , PSO, and GA when these algorithms are used to handle time-consuming problems[25][26][27]. These papers concluded that DE performs better than others. DE converges faster and can usually get better results.

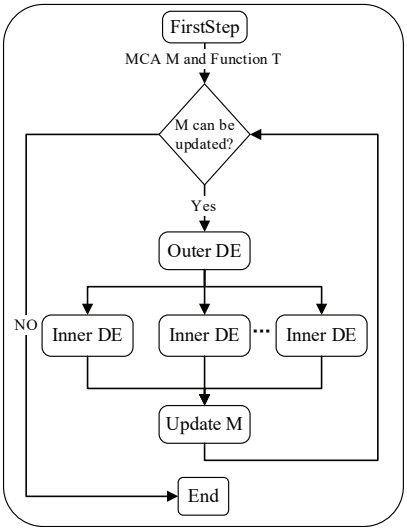


Fig. 2: Algorithm workflow.

Compared with classical optimization methods, DE also have advantages. It can handle problems which have large search spaces. Besides, it can be used for multidimensional real-valued functions but does not use the gradient of the problem being optimized, which means DE does not require for the optimization problem to be differentiable as is required by classic optimization methods such as gradient descent and quasi-newton methods. DE can, therefore, be used on optimization problems that are not even continuous[28]. Finding time-optimal MCAs is an optimization problem. The search space can be very large and the testing time function may not be differentiable, thus we apply DE algorithm to solve it.

We have got the size-aware solution $M$and the testing time function T, the second step is ready to be started. In this step, some rows in M might be replaced by more other rows while the new MCA's total testing time decreases. The workflow of our NDE algorithm is shown in Fig. 2. If we find M can be updated, which means M exists at least one row that can be replaced, we use the outer layer algorithm to determine the number of rows used to replace that row and we use the inner layer algorithm to find each of these rows.

Let r_i be the ith row of M, then r_i contains some t-tuples that are unique, which means these t-tuples just appears once in M. So only r_i contains these t-tuples and thus we call these t-tuples the $unique\ t\text{-}tuples$ of r_i. The number of unique t-tuples that r_i contains is represented by u_i. If there are some rows that can replace r_i, then these rows must have the following two properties:

1) They must contain each unique t-tuple at least once.

2) The total testing time of them is less than that of r_i.

To verify if r_i can be replaced by m other rows, we should distribute the u_i unique t-tuples to m rows, we call this a $distribution$. Under each distribution, we can find m rows with the least testing time with the help of DE in the inner layer. DE in the outer layer can find the distribution which decreases the testing time of r most. The details of our algorithm are shown below.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

Algorithm 2 Inner_Differential_Evolution(set TU)

```
procedure IDE
    Initiate(IM, TU)
    gNum_inner=0
    while !stopcriterion do
        for i=1 to IN do
            mutation operation
            crossover operation
            selesction operation
        end for
        get newIM
        IM ← newIM
        gNum_inner++
    end while
    find the best agent r' in IM
    return row r'
end procedure
```

A. DE in the inner layer

Our method takes rows as units and let r be the original row to be replaced. Suppose the components of r are $(p_1, p_2, ..., p_k)$ and the number of unique t-tuples of r is u_r. Now we are trying to replace r with m other rows and r' is one of the m rows. The number of unique t-tuples r' has is $u_{r'}$. Each tuple is composed of t components. The components compose these $u_{r'}$ tuples are fixed while other components are $undecidable$. So the number of all the possibilities of r' is at least $p = \prod_{i=1}^{d} v_{c_i}$ while d is the number of undecidable components and the c_ith component of r' is undecidable. Our task is to find r' with the least testing time in such a large space. We can get the function for calculating the testing time of each test case through the ways we said in 3.2. Let T be the function and the testing time of r' is $T(r') = T(p'_1, p'_2, ..., p'_k)$. T is a function with discrete variables and thus classic optimization methods can not be applied to find the best solution of T. Fortunately, DE does not require for the problem to be differentiable. So DE can be applied for finding the best r'. Standard DE contains 4 steps: initial population, mutation operation, crossover operation and selection operation. We apply standard DE to the problem of finding r'. Algorithm 2 shows the pseudocode of DE in the inner layer and the details of each stage are shown below. Parameter TU is the set of unique t-tuples of r' and is used to initialize the search space IM which has IN agents. $gNum_{inner}$ is the current generation.

1) Initial population: There are IN agents $(x_1, x_2, ..., x_{IN})$ in the search space IM. Each agent has k chromosomes and is a test case. So agent x_i is denoted by a vector $(x_{i1}, x_{i2}, ..., x_{ik})$, in which x_{ij} represents the jth chromosome of the ith agent. The possible values x_{ij} can have must belong to the set S_j.

We should initialize the IN agents. Each agent is a test case. When generating an agent, there are two constraints: First, the components that are decided by the unique t-tuples r' has should be same as these in r. Second, the jth value must come from the set $S_j (1 \leq j \leq k)$. Note that each time we reuse DE for a new r' the IN agents are generated again.

2) Mutation operation: For each agent $x_i(g)$ of the gth generation, we pick three agents $x_{r_1}(g), x_{r_2}(g)$ and $x_{r_3}(g)$ from the IN agents at random. These four agents must be distinct from each other, which means $i \neq r_1 \neq r_2 \neq r_3$. Then we compute a new candidate agent y of the next generation as follows: $y_i(g+1) = x_{r_1 i}(g) + F(x_{r_2 i}(g) - x_{r_3 i}(g))$. F is the scale factor. The value of F is important because it is critical that whether the optimization problem can get the global optimal solution or can only get the local optimal solution. F is determined by a formula. Let $\lambda = e^{1-gNum/(gNum+1-g)}$ and then $F = F_0 * 2^{\lambda}$. F_0 is the initial value of F and $gNum$ is the maximum generation while g is the current generation.

Besides, y_i should belong to the set S_i. However, y_i is not an integer in most situations. Changing y_i to the nearest value $\lfloor y_i \rfloor$ in the set S_i is not a good strategy. That might lead y_i being a specific value and does not change anymore, which means we can only get a local optimal solution. To solve the problem, we propose a new method. If y_i is not an integer, then there is a probability $PI = |y_i - \lfloor y_i \rfloor|$ that y_i is set to $\lceil y_i \rceil$, otherwise y_i is set to $\lfloor y_i \rfloor$.

3) Crossover operation: From the mutation operation we get a candidate agent $y_i(g+1)$. Now we can get a new candidate agent $z_i(g+1)$ by crossover operation. Note that $z_i(g+1)$ should contain all the unique t-tuples of $x_i(g)$. Thus some components of $z_i(g+1)$ are fixed. The undecidable components of $z_i(g+1)$ are determined by (3)

$$\mathrm{z_{ij}(g+1)} = \begin{cases} y_{ij}(g+1), & if\ rand(0,1) \leq CR \\ & or\ j = j_{rand} \\ x_{ij}(g), & otherwise \end{cases} \tag{3}$$

Note that the jth component of $z_i(g+1)$ is undecidable. CR is crossover probability. j_{rand} is a random integer between 1 and k and the j_{rand}th component of r' is undecidable. If we can't find such a j_{rand}, then all the components of r' are fixed because of the unique t-tuples r' has and thus the DE should be terminated. The existence of j_{rand} guarantees that at least one chromosome of $z_i(g+1)$ comes from y_i. So $z_i(g+1)$ has changed compares to $x_i(g)$.

4) Selection operation: We use a greedy method to choose proper agents for next generation.

$$\mathrm{x_i(g+1)} = \begin{cases} z_i(g+1), & if\ T(z_i(g+1)) < T(x_i(g)) \\ x_i(g), & otherwise \end{cases} \tag{4}$$

We use (4) to select the final $x_i(g+1)$. This greedy choice is very important because it guarantees that the testing time of r' decreases. So the total testing time of the m rows which are used to replace r becomes less and less. Note that T is the function for calculating the testing time of each test case.

Mutation, Crossover, and Selection operation should run IN times so that the IN agents in IM are improved.

5) Termination Condition: If the algorithm terminates, we can get the best solution according to the function T in the latest generation. This solution is assigned to r'.

The terminal conditions of our algorithm are listed below:

1) Such a j_{rand} does not exist.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

2) The best agent does not change in $gChg$ generations.
3) The largest number of generation $gNum$ is reached.

B. DE in the outer layer

If r can be replaced by n other rows (the n rows should have the two properties mentioned above), then we say that r has the $degree$ of n. To verify if r has the degree of m, we should first distribute its u_r unique t-tuples to m rows and we call this a $distribution$. Based on each distribution, we can get at most m rows. If the total testing time of the m rows is less than that of r, it means the m rows (or the distribution) can decrease the testing time of MCA and thus can be used to replace r. Different partitions of the u_r tuples can lead to different distributions. There are m^{u_r-1} possible distributions and DE in the outer layer can find the distribution which decreases the testing time of MCA most. Note that m can be set to proper values (from 1 to u_r) based on our needs. The bigger m is, the longer time NDE needs.

We use a vector $d = (d_1, d_2, ..., d_{u_r})$ to represent a distribution. d_i is mapped to the ith unique t-tuple of r. The possible values d_i (i = 1,2, ... ,u_r) can get from are 0 to m-1. This set of values is denoted as DS_m. All the components $(d_1, d_2, ...)$ of d are divided into m groups according to the components' values. Components having the same values are in the same group. For example, if d_1 and d_2 are both equals to 1, then they are put into the same group. d_i is mapped to the ith unique t-tuple of r. So if d_i equals to g, then the gth row should contain the ith unique t-tuple of r. Using this strategy, the u_r unique t-tuples are divided into m groups and each group is used to construct one new row. This new row should cover all the unique tuples that are mapped to the components in this group. When constructing such a row (this row can be considered as r' mentioned in IV.A), DE in the inner layer can be used to find the one with the least testing time.

Suppose we get m rows $(r_1, r_2, ..., r_m)$ from the m groups. Then the testing time of the m rows is $G(r) = T(r_1)+T(r_2)+ ... + T(r_m)$. Different distributions can lead to different m rows and thus different testing time. So we use DE to find the best distribution. Best means under this distribution the testing time of the m rows is the least. DE in the inner layer can be applied when finding one of the m rows. Algorithm 3 shows the pseudocode of DE in the outer layer. The parameter m indicates that we try to replace r with at most m rows. The search space is OM which has ON agents. $gNum_{outer}$ is the current generation. The details of DE in the outer layer are shown below.

1) Initial population: There are ON agents $(x_1, x_2, ..., x_{ON})$ in the search space OM. Each agent has u_r chromosomes and is a distribution. So agent x_i is denoted by a vector $(x_{i1}, x_{i2}, ..., x_{iu_r})$, in which x_{ij} represents the jth chromosome of the ith agent. The possible values x_{ij} can have must belong to the set DS_m.

We should initialize the ON agents with the constraint that each component's value of an agent must come from the set DS_m. Note that each time we reuse DE for a different degree, the set DS_m should be changed too.

2) Mutation operation: For each agent $x_i(g)$ of the gth generation, we pick three agents $x_{r_1}(g), x_{r_2}(g)$ and $x_{r_3}(g)$ from the ON agents at random. These four agents must be distinct from each other, which means $i \neq r_1 \neq r_2 \neq r_3$. Then we compute a new candidate agent y of the next generation as follows: $y_i(g+1) = x_{r_1 i}(g) + F(x_{r_2 i}(g) - x_{r_3 i}(g))$. F is a new scale factor that is different from that in the inner layer DE. F is important because it is critical that whether the optimization problem can get the global optimal solution or can only get the local optimal solution. F is also calculated by the formula mentioned above.

Besides, y_i should belong to the set DS_m. However, y_i is not an integer in most situations. As we have said, changing y_i to the nearest value $\lfloor y_i \rfloor$ in the set DS_m is not a good strategy. That might lead y_i being a specific value and does not change anymore, which means we can only get a local optimal solution. To solve the problem, we use the method we have introduced. If y_i is not an integer, then there is a probability $PI = |y_i - \lfloor y_i \rfloor|$ that y_i is set to $\lceil y_i \rceil$, otherwise y_i is set to $\lfloor y_i \rfloor$.

3) Crossover operation: From the mutation operation we get a candidate agent $y_i(g+1)$. Now we can get a new candidate agent $z_i(g+1)$ by crossover operation. $z_{ij}(g+1)$ is the jth chromosome of $z_i(g+1)$ and is determined by (5)

$$\mathrm{z_{ij}(g+1)} = \begin{cases} y_{ij}(g+1), & if\ rand(0,1) \leq CR \\ & or\ j = j_{rand} \\ x_{ij}(g), & otherwise \end{cases} \quad (5)$$

Note that CR is a new crossover probability. j_{rand} is a random integer between 1 and u_r . There is not any constraint to j_{rand} . The existence of j_{rand} guarantees that $z_i(g+1)$ has changed compares to $x_i(g)$.

4) Selection operation: We use a greedy method to choose proper agents for next generation.

$$\mathrm{x_i(g+1)} = \begin{cases} z_i(g+1), & if\ G(z_i(g+1)) < G(x_i(g)) \\ x_i(g), & otherwise \end{cases} \quad (6)$$

We use (6) to select the final $x_i(g+1)$. This greedy method guarantees that the total testing time of the m rows becomes less and less. Note that if x is an agent, we can divide the unique t-tuples of x into m groups and thus we can get m rows. Then DE in the inner layer can be used to find the best solution of each row. Summing all the testing time of these rows we can get $G(x)$.

Mutation, Crossover, and Selection operation should run ON times so that the ON agents in OM are improved.

5) Termination Condition: If the algorithm terminates, we can find the best solution d_{best} in the latest generation. Then We can get m rows to replace r according to d_{best}.

The terminal conditions of our algorithm are listed below:
1) The best agent does not change in $gChg$ generations.
2) The largest number of generation $gNum$ is reached.

Note that $gChg, gNum$ are different from those in the inner layer DE.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

Algorithm 3 Outer_Differential_Evolution (integer m)

```
1: procedure ODE
2:     Initiate(OM)
3:     gNum_outer=0
4:     while !stopcriterion do
5:         for i=1 to ON do
6:             mutation operation
7:             crossover operation, get new agent new_d_i
8:             m rows ← m groups ← agent new_d_i
9:             for i=1 to m do
10:                get unique t-tuple set tu_i
11:                find the best r_i through Inner_DE(tu_i)
12:            end for
13:            Calculate G(new_d_i) = T(r_1) + ... + T(r_m)
14:            Select between d_i and new_d_i use function G
15:        end for
16:        get newOM
17:        OM ← newOM
18:        gNum_outer++
19:    end while
20:    find the best agent D in OM
21:    return distribution D
22: end procedure
```

Algorithm 4 Nested_Differential_Evolution (MCA M)

```
1: procedure NDE
2:     while (There exists one row r can be replaced) do
3:         select r from M
4:         set initial solution s_0 to r
5:         calculate u_r, T(r)
6:         set degree d=1
7:         while d < u_r do
8:             find d rows to replace r
9:             use Outer_DE(d), get a distribution D
10:            d rows ← D
11:            if T(s_0)>T(d rows) then
12:                s_0 ← d rows
13:                d++
14:            end if
15:        end while
16:        update M, replace r with s_0
17:    end while
18: end procedure
```

C. The combination of these two kinds of DE

Since we have presented two kinds of DE algorithm, the last mission is to combine them in order to find cost-aware MCAs.

The pseudocode of our NDE algorithm is shown in Algorithm 4. The steps are listed below:

1) At the beginning, row r of M is chosen. Solution s_0 is used to replace r and is initiated to r.

2) Next, we calculate u_r and $T(r)$.

3) Then we try to replace r with at most 1,2, ... ,u_r rows using DE in the outer layer. If we find some rows that can replace s_0 and the total testing time is less than that of s_0, then s_0 is updated to these rows.

4) When the process ends, we update M by replacing r with s_0.

5) Repeat step 1 to 4 until we can not find one row that can be replaced.

D. An example

To make our NDE algorithm clear, we give an example in this section. Suppose there is a system which has 3 parameters and each parameter has 2 possible values. A possible strength-2 MCA is shown in Figure. 3.

The first row r_0 contains three unique 2-tuples. They are 00x,0x1,and x01 (00x means the 2-tuple is 00 in the first and second columns). So r_0 can be replaced by at most three other rows with each of them contains one unique 2-tuple of r_0. However, r_0 can also be replaced by two rows. The three unique 2-tuples can be distributed in three ways. One new row contains one 2-tuple and the other new row contains two 2-tuples. So there is a total of four possible distributions. DE in the outer layer is used to determine which distribution can get the least testing time. In this process, we must get each distribution's testing time. For example, suppose the current distribution is replacing r_0 with three rows. Each of the three rows has a unique 2-tuple. So the three rows are 00*,0*1,*01 (* means this component can be every possible value). DE in the inner layer can be used to find the best 00* and the other two rows. Then we can get the testing time of this distribution. Other distributions' testing time are calculated in the same way. So DE in the outer layer can find the best distribution based on DE in the inner layer. That is how DE in the inner layer and DE in the outer layer cooperates. Finally, we can get some rows to replace r_0.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Fig. 3: A possible MCA.

V. Experimental results

This section presents the experimental results of our algorithm. Our experiment contains two parts. The first part intends to demonstrate that DE in the outer layer is necessary and we will study how to make DE in the outer layer has the best performance. Then our NDE algorithm is compared with some state-of-the-art algorithms using some commonly used benchmarks. In the second part, we use NDE to construct MCAs for some configurable software. A program analysis tool CPAchecker, a compression tool pigz, and a build automation tool Make. MCAs generated by NDE for these tools should require less testing effort compared with those generated by other methods.

The first part of our experiment is presented just to demonstrate that our algorithm performs quite better for some mathematical model. Our algorithm also works well for real-world

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

applications and we will justify that in the second part. All the experiments are run on a CPU Intel(R) Core(TM) i7-4790K @ 4.00GHz, 8GB of RAM with Ubuntu16.04LTS. The following parameters are used in our experiments:

1) $gNum_{inner}$ = 50, $gNum_{outer}$ = 15
2) IN_{inner} = 50, ON_{outer} = 15
3) F_{0inner} = F_{0outer} = 0.2
4) CR_{inner} = CR_{outer} = 0.2
5) $gChg_{inner} = gChg_{outer}$ = 10

Note that m is set to 2 in all the experiments except the part $Influence\ of\ DE\ in\ the\ outer\ layer$. It means that one row can be replaced by at most two other rows. With that setting, our algorithm can have a competitive speed and we can get satisfactory results.

The values of the parameters have a great impact on the performance of evolutionary algorithms. Thomas Stutzle and his group[29][30], Gusz Eiben and his group[31][32] have proposed some useful algorithms for tuning parameters. Thus using some existing methods to calibrate our algorithm's parameters is a meaningful work in the future.

A. The effectiveness of our NDE algorithm

1) Influence of DE in the outer layer: In our NDE algorithm, DE in the outer layer is used to replace one row by at most m rows while DE in the inner layer is used to find one of them. In this experiment, we will show that DE in the outer layer is necessary, which means sometimes replace one row by more other rows is better than by just one row.

We try to generate four MCAs as shown in Table III. We use four strategies to get MCAs. Considering that MCA's size is different when different strategies are applied, the sizes of MCAs are represented by *.

(1) An MCA is generated using the SA algorithm mentioned in 3.1 and we use M to represent the result. Obviously, M has the optimal or near optimal size.

(2) DE in the inner layer is applied to update each row of M with another row.

(3) DE in the outer layer is applied to update each row of M with at most two other rows.

(4) DE in the outer layer is applied to update each row of M with at most three other rows.

Note that the new rows in strategy 2, 3 and 4 must satisfy the two constraints described in section IV. If we update M through trying to replace each row by at most m rows. m is called the $cardinality$ of this new MCA. Note that the cardinality of new MCA is 0 if it is the same with M, which means M has not been updated.

The results of the four strategies are illustrated in Fig. 4 and Fig. 5. Both Fig. 4 and Fig. 5 use the MCAs in Table III. We assume the real testing time functions are $f_1 = x_1 * x_2 + x_2 * x_3 + ...x_{k-1} * x_k$ (hours) and $f_2 = (x_1 + x_2 + ...x_k)^2$. Then we can get some data about configurations and their corresponding testing time. We use these data to fit two new functions. The new functions are used in Fig. 4 and Fig. 5, respectively. Finally, f_1 and f_2 are used to evaluate the results. In each pair of figures in Fig. 4 and Fig. 5, the left fig's plot represents the testing time of the MCA with different cardinalities and the right fig's plot represents the size of the MCA with different cardinalities. Different cardinalities are distinguished by the color of histograms. Fig. 4 and Fig. 5 disclose the fact that the total testing time of MCAs has nothing with their sizes and DE in the outer layer is very effective. So an MCA with a small size can be verified if its testing time can be reduced by replacing some rows by other rows. The two figures also show that the larger cardinality is, the better MCA we can get. In a word, if we replace one row by at most m (m=2, 3, 4, ...) rows we can get a better result than by just one row.

2) The NDE's performance: In this section, we will study how to make DE in the outer layer has the best performance. We mainly focus on the size of the search space and the largest number of generation (ON and $gNum$).

In this experiment, we try to generate three MCAs shown in Table IV. The experiment contains two parts:

(1) We set ON to 15 and $gNum$ changes from 5 to 25.

(2) We set $gNum$ to 15 and ON changes from 5 to 25.

The results of the experiment are illustrated in Fig. 6 and Fig. 7. $f_1 = x_1 * x_2 + x_2 * x_3 + ...x_{k-1} * x_k$ (hours) is assumed to be the real testing time function. In Fig. 6 the fig's plot represents the testing time of the MCA against the value of ON and $gNum$ with $gNum$ varies from 5 to 25 and in Fig. 7 the fig's plot represents the testing time of the MCA against the value of ON and $gNum$ with ON varies from 5 to 25. Fig. 6 and Fig. 7 disclose that the MCA's testing time decreases with the growth of $gNum(ON)$. However, there are few changes when $gNum$ and ON are large enough.

3) Comparing our algorithm with some state-of-the-art algorithms: In this section, our NDE algorithm is compared with some state-of-the-art algorithms such as SA [3], TCG [18], ACTS [19], AllPairs [33], Jenny [34],ProTest [35] and TestCover [36]. They can construct MCA and we focus on the MCA's total testing time. Of course, MCA which has the least testing time is the best. $f_1 = x_1 * x_2 + x_2 * x_3 + ...x_{k-1} * x_k$ (hours) is assumed to be the testing time function in this experiment.

The results are shown in Table V. The first column shows IDs of MCAs using in this experiment and these IDs are also used in Fig. 8. The second column is the benchmark. Column 3 to 10 describes the testing time of MCAs generated by different methods. The last two columns show the results

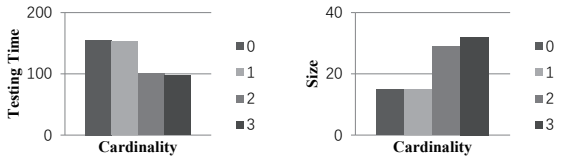
TABLE III: Set of four MCAs

ID	Mixed Covering Array description
mca_0	MCA(*;2,11,$5^1 3^8 2^2$)
mca_1	MCA(*;2,19,$6^1 5^1 4^6 3^8 2^3$)
mca_2	MCA(*;2,21,$5^1 4^4 3^{11} 2^5$)
mca_3	MCA(*;3,13,$5^1 4^1 3^2 2^9$)

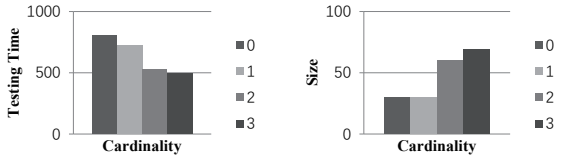
TABLE IV: Set of three MCAs

ID	Mixed Covering Array description
mca_0	MCA(*;2,11,$5^1 3^8 2^2$)
mca_1	MCA(*;2,19,$6^1 5^1 4^6 3^8 2^3$)
mca_2	MCA(*;3,13,$5^1 4^1 3^2 2^9$)

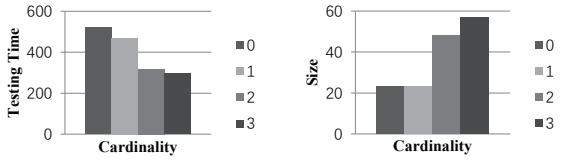
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation



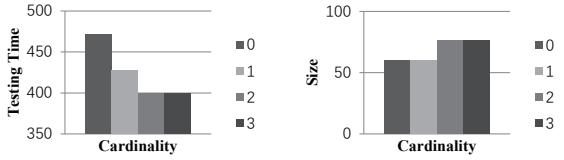
(a) MCA(*;2,11,$5^1 3^8 2^2$)



(b) MCA(*;2,19,$6^1 5^1 4^6 3^8 2^3$)

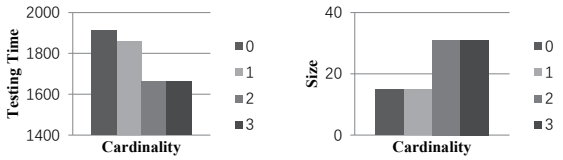


(c) MCA(*;2,21,$5^1 4^4 3^{11} 2^5$)

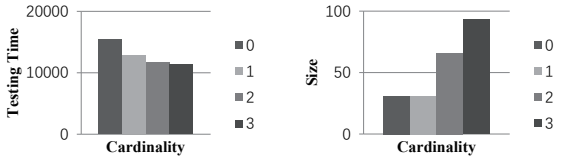


(d) MCA(*;3,13,$5^1 4^1 3^2 2^9$)

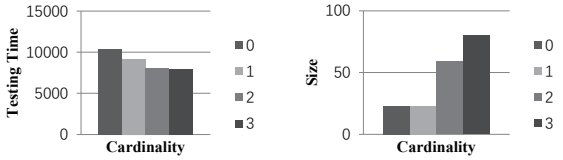
Fig. 4: The testing time and size of MCAs with different cardinalities and the testing time function is f_1.



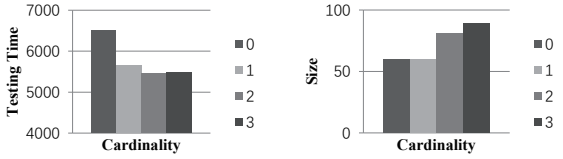
(a) MCA(*;2,11,$5^1 3^8 2^2$)



(b) MCA(*;2,19,$6^1 5^1 4^6 3^8 2^3$)



(c) MCA(*;2,21,$5^1 4^4 3^{11} 2^5$)



(d) MCA(*;3,13,$5^1 4^1 3^2 2^9$)

Fig. 5: The testing time and size of MCAs with different cardinalities and the testing time function is f_2.

of our NDE algorithm and the best-reported results. It can be seen that our method has the best performance, which implies that our method can generate cost-aware MCAs.

Fig. 8 compares these methods by broken lines. The plot represents the MCAs (abscissa) against the testing time attained by the compared procedures (ordinate). It is obvious that our NDE algorithm performs better than other methods because the broken line representing NDE is in the bottom, which is to say, the MCAs generated by NDE has the least testing time.

B. Constructing cost-aware MCAs for real-world applications

This experiment is proposed to demonstrate that our NDE algorithm can be used to generate cost-aware MCAs for real-world applications. We used CPAchecker, pigz and Make.

1) CPAchecker: CPAchecker is a configurable tool for software verification. It took part in the 4th Intl. Competition on Software Verification held at TACAS 2015 in London, UK and performed well. CPAchecker can be configured to perform not only a purely tree-based or a purely lattice-based analysis, but offers many intermediate settings[37]. CPAchecker can use many abstract interpreters, such as an apron analysis, a predicate abstraction, a shape analysis and so on. You can get more information about CPAchecker from https://cpachecker.sosy-lab.org/ . Each kind of CPA can be configured using several parameters and under each configuration these interpreters should verify software correctly.

In our experiment, we first pick 11 parameters to configure CPA. However, according to the work of Myra B. Cohen and his group, the combinatorial interaction testing (CIT) problems can be constrained. If a parameter can be tested alone we do not need to consider it in CIT models so that we can reduce the test space [38] [39] [40] [41]. Thus we simplify CPAchecker's model to have only 8 parameters based on this technique. The model is MCA(*;2,8,$3^2 2^6$). The parameters and the size of the set of their possible values are shown in Table VI.

We intend to generate a test suite of configurations for a predicate analysis CPA and use an MCA to represent the test suite. The MCA's strength is 2 and its testing time should less than that of MCAs generated by other tools. We use the benchmark sv-benchmarks of the competition mentioned above to test CPA. Considering the fact that there are thousands of programs in the benchmark and our purpose is just to state the effectiveness of our NDE algorithm instead of really test CPA, we select one program named $linux-4.2-rc1.tar.xz-08_1a-drivers--ata--libata.ko-entry_point_true-unreach-call.cil.out.c$ in directory c/ldv-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

TABLE V: Results of our method and some state-of-the-art algorithms

ID	MCA description	ACTS	AllPairs	Jenny	Pict	ProTest	TestCover	SA	Our Method	Best Reported Results
1	$t2k11v5^1 3^8 2^2$	158	219	209	186	192	224	154	112	112
2	$t2k9v4^5 3^4$	295	318	350	345	310	321	272	219	219
3	$t2k21v5^1 4^4 3^{11} 2^5$	516	718	739	718	736	1054	528	319	319
4	$t2k19v6^1 5^1 4^6 3^8 2^3$	877	1155	1212	1040	1192	1870	813	531	531
5	$t2k17v6^2 4^6 2^9$	955	1131	979	925	994	1314	819	673	673
6	$t2k19v7^1 6^1 5^1 4^5 3^8 2^3$	1248	1849	1755	1558	1703	2401	1257	871	871
7	$t2k14v6^3 4^8 2^3$	1085	1703	1669	1623	1611	2007	1187	879	879
8	$t2k16v6^4 4^5 2^7$	1379	1765	1743	1926	1776	2170	1241	1015	1015
9	$t2k15v6^5 4^3 2^7$	1762	1902	1855	2029	2007	2346	1429	1219	1219
10	$t2k14v6^5 5^5 3^4$	2075	2964	2930	3151	3060	3792	2971	1531	1531
11	$t2k8v8^2 7^2 6^2 5^2$	3881	4256	4195	4486	4259	4271	3486	3233	3233
12	$t3k6v5^2 4^2 3^2$	1324	-	1612	1466	-	-	1175	1146	1146
13	$t3k6v6^2 4^3 3^1$	2501	-	2987	2900	-	-	2318	2305	2305
14	$t3k7v4^3 3^4$	754	-	777	795	-	-	601	570	570
15	$t3k12v4^3 3^2 2^7$	642	-	767	771	-	-	579	520	520

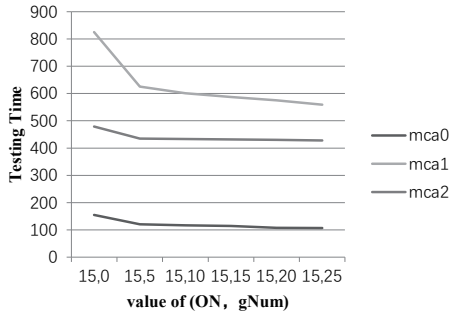


Fig. 6: The testing time of MCAs with $gNum$'s value varies from 5 to 25.

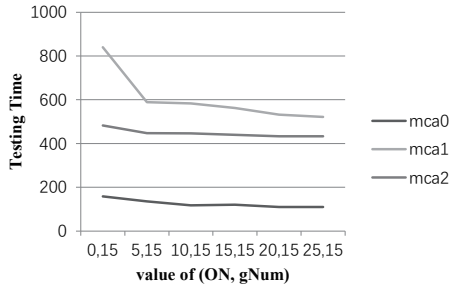


Fig. 7: The testing time of MCAs with ON's value varies from 5 to 25.

linux-4.2-rc1 to test CPA.

Under each configuration, the predicate analysis CPA run this program and the running time is treated as the testing time of this configuration. The testing time varies from 10s to 800s and thus our NDE can be applied. We run the program in 100 different configurations. Since we do not need the testing time function to be accurate. We can guess the pattern of testing time function based on the sample test cases empirically. Then we can use Matlab to fit the function using linear regression method. For example, if f' is proportional to (1-x) and (1-y), we can guess the function's pattern is $m*(1-x)+n*(1-y)$ and use linear regression to get m and n. In that way, we get the testing time function $T(x) = 6.7713 - 5.5377 * x_4 + (1 - x_4)(840.68 * x_5 - 835.51 * x_5 * x_8 - 102.27 * (1 - x_8)(x_1 - 1) * x_1 * x_5)$ (s). Note that x is an eight-dimensional vector and represents a configuration. Using the definition of $\delta(f, f')$, C and ε in section III.B, we can get δ = 17.86 and $C\varepsilon$ = 118.5. The requirements in section III.B are satisfied. Then we use our NDE algorithm to get cost-aware test-suites. Our algorithm's result are compared with those obtained by ACTS and SA. MCA generated by ACTS has 10 rows and the other two MCAs both have 9 rows. The testing time of each MCA is shown in Table IX.

In Table IX, CTime represents the MCA's testing time obtained through the testing time function T while RTime is the MCA's real testing time. BestN shows the best result obtained among ACTS and SA. Best is the least testing time obtained among the three methods and DP (Decline Percentage) shows the percentage of MCA's reducing testing time by our NDE algorithm.

2) pigz: pigz is a parallel implementation of gzip for modern multi-processor and multi-core machines. We use pigz(2.3.4) in our experiment. It has many parameters and we use the technique mentioned above to eliminate parameters that can be tested alone for the purpose of reducing test space. Finally, the model has only 9 parameters. The model can be represented by MCA(*;2,9,$3^2 2^7$) and the parameters we used

TABLE VI: Configurable options of CPA

ID	Configurable Options	Value Set Size
p_1	analysis.reachedSet	3
p_2	analysis.traversal.order	3
p_3	analysis.machineModel	2
p_4	parser.dialect	2
p_5	analysis.interprocedural	2
p_6	analysis.stopAfterError	2
p_7	output.disable	2
p_8	analysis.functionPointerCalls	2

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2017.2747638, IEEE Transactions on Evolutionary Computation

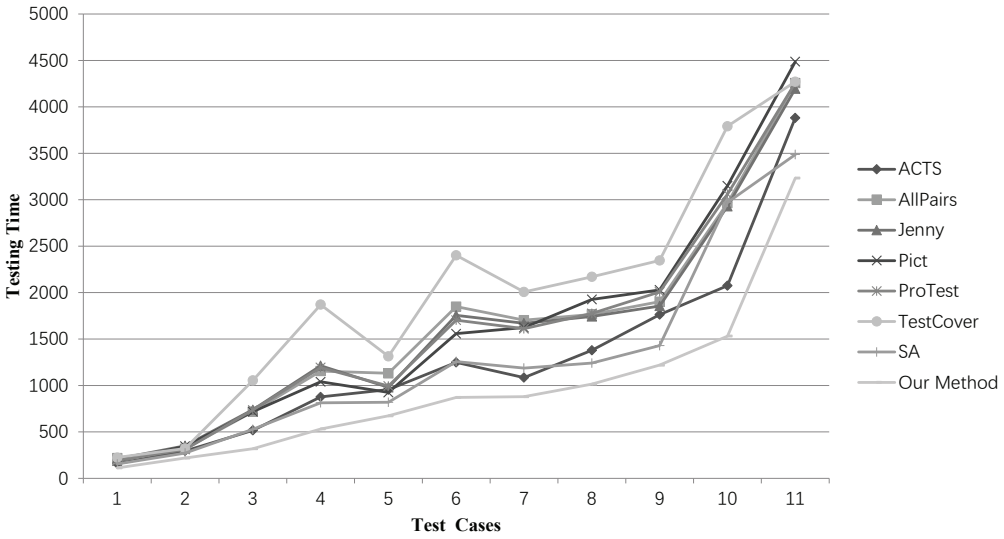


Fig. 8: Graphical comparison of the testing time of MCAs found by our method with respect to those produced by some state-of-the-art algorithms.

in our experiment are shown in VII. The first parameter is compression level, it varies from 1 to 9. The second parameter is the number of threads. For simplicity, we only use three levels 1,5,9 of the first parameter and 1,4,8 threads of the second parameter. The others are flags and both of them has two values(be used or not).

We use pigz to compress a directory which has videos, audios, pictures and text files. The directory's size is 1G. Under each configuration, the compression time is treated as the testing time of this configuration. The testing time varies from 7s to 180s and thus we can use NDE to get cost-aware test-suite. We run 100 different configurations to get the testing function. Using the method we have said in CPAchecker part, we get the testing time function $T(x) = 11.5286 + 1.3594 * x_1 + 147.8646 * x_3 - 0.9844 * x_9 + (1 - x_3)(-2.1510 * x_2 + 8.9844 * (x_2 - 1)(x_2 - 2))$ (s). Note that x is a nine-dimensional vector and represents a configuration. We get δ = 12.88 and $C\varepsilon$ = 21.95. The requirements are satisfied. Then our NDE algorithm can be applied. Our algorithm's result are compared with those obtained by ACTS and SA. MCA generated by ACTS has 11 rows and the other two MCAs both have 9 rows. The testing time of each MCA is shown in Table IX.

3) Make: We also use GNU Make(4.1), a tool that can build executable programs from source code. It has many parameters. So the test space is too large. We use the technique introduced by Myra B. Cohen and his group[38] [39] [40] [41] to reduce test space. The parameters which can be tested alone can be removed. After applying the technique the model of Make only has 11 parameters. Make's model is MCA(*;2,11,$3^2 2^9$) and the parameters are shown in VIII. The first parameter is the number of jobs and its value can be any number. The second parameter is the load bound and its value can be any real value. For simplicity, the number of jobs in our experiment is 1,4,8 and the value of load is 1.0 , 2.5 , 4.0 . The others have two values (be used or not).

We use Make to build Vim, a well-known text editor. The time used for building Vim is treated as the testing time. According to our results, the testing time varies from 10s to 70s. We run Make in 100 different configurations and record the testing time under each configuration. Then we used the method we used in the part CPAchecker to get the testing time function $T(x) = 47.4920 - 37.3479 * x_4(1 - x_6)(1 - x_7) +$

TABLE VII: Configurable options of pigz

ID	Parameter	Explanation	Value Set Size
p_1	-1 to -9	compression level	3
p_2	-$p[N]$	compression threads	3
p_3	-c	write output to stdout	2
p_4	-i	block independent	2
p_5	-K	keep origin file	2
p_6	-q	print no message	2
p_7	-R	rsyncable	2
p_8	-v	verbose output	2
p_9	-z	compress to zlib	2

TABLE VIII: Configurable options of Make

ID	Parameter	Explanation	Value Set Size
p_1	-$j[N]$	allow N jobs	3
p_2	-$l[N]$	load bound	3
p_3	-b	ignored compatibility	2
p_4	-B	unconditional make	2
p_5	-d	print debug info	2
p_6	-i	ignore errors	2
p_7	-k	keep going	2
p_8	-p	print database	2
p_9	-r	no built-in rules	2
p_{10}	-R	no built-in variables	2
p_{11}	-s	don't echo recipes	2

TABLE IX: Testing time of MCAs generated by different methods

Application		ACTS	SA	NDE	BestN	Best	DP
CPA	CTime	1722	885	680	885	680	23.16
	RTime	1727	871	683	871	683	21.58
pigz	CTime	1174	875	573	875	573	34.51
	RTime	1161	899	598	899	598	34.59
Make	CTime	499	331	257	331	257	22.36
	RTime	431	323	259	323	259	19.81

$\lceil((1-x_4+x_6+x_7)/3)\rceil * (-0.5282 * x_1(1-x_2)(1-x_3) - 28.0028 * x_1 * x_2)$ (s). x is an eleven-dimensional vector and represents a configuration. We get δ = 7.32 and $C\varepsilon$ = 9. The requirements are satisfied. MCA generated by ACTS has 12 rows and the other two MCAs both have 9 rows. The testing time of each MCA is shown in Table IX.

4) results: The results in Table IX indicate that our algorithm can be used to generate cost-aware MCAs for real-world applications. Comparing with representative state-of-the-art algorithms, our NDE algorithm can reduce testing time by more than 20 percents. If we use more complicated programs for CPAchecker as input, bigger files for pigz to compress, or larger projects for Make to build, the gap will become larger. Note that the testing time function is fitted so the results may not be very accurate. If we use NDE in regression testing phase we can get accurate testing time and the results may be more positive.

VI. Conclusions

This paper focuses on how to construct cost-aware MCAs using NDE algorithm with the initial solution provided by SA algorithm. Our method contains two steps:

1) An existing SA algorithm is used to find a size-aware MCA. Then the result is provided to the second step as the initial solution.

2) In order to get a cost-aware MCA, a novel NDE algorithm is proposed to improve the MCA obtained in the first step.

Our NDE algorithm mainly contains two components, DE in the inner layer and DE in the outer layer. The former is used to replace the old row by another row, while the latter is used to find the best distribution. The experimental results imply that our method is efficient and is better than some state-of-the-art algorithms in finding cost-aware MCAs.

Acknowledgement

This research is sponsored in part by NSFC Program (No. 91218302, No. 61527812, 61402248), National Science and Technology Major Project (No. 2016ZX01038101), MIIT IT funds (Research and application of TCN key technologies) of China, and The National Key Technology R&D Program (No. 2015BAG14B01-02).

The authors would like to thank Xinrui Guo and Zonghui Li for their valuable suggestions.

References

[1] Y. Jiang, H. Zhang, X. Song, X. Jiao, W. N. Hung, M. Gu, and J. Sun, "Bayesian-network-based reliability analysis of plc systems," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 11, pp. 5325–5336, 2013.

[2] Y. Jiang, H. Zhang, H. Liu, W. Hung, X. Song, M. Gu, and J. Sun, "System reliability calculation based on the run-time analysis of ladder program," *IEEE Transactions on Industrial Electronics*, 2014.

[3] H. Avila-George, J. Torres-Jimenez, L. Gonzalez-Hernandez, and V. Hernández, "Metaheuristic approach for constructing functional test-suites," *Software, IET*, vol. 7, no. 2, pp. 104–117, 2013.

[4] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pp. 91–95, IEEE, 2002.

[5] D. R. Kuhn, D. R. Wallace, and A. M. Gallo Jr, "Software fault interactions and implications for software testing," *Software Engineering, IEEE Transactions on*, vol. 30, no. 6, pp. 418–421, 2004.

[6] C. J. Colbourn, M. B. Cohen, and R. Turban, "A deterministic density algorithm for pairwise interaction coverage.," in *IASTED Conf. on Software Engineering*, pp. 345–352, Citeseer, 2004.

[7] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete applied mathematics*, vol. 138, no. 1, pp. 143–152, 2004.

[8] G. Seroussi and N. H. Bshouty, "Vector sets for exhaustive testing of logic circuits," *IEEE Transactions on Information Theory*, vol. 34, no. 3, pp. 513–522, 1988.

[9] Y. Jiang, H. Zhang, H. Zhang, H. Liu, X. Song, M. Gu, and J. Sun, "Design of mixed synchronous/asynchronous systems with multiple clocks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2220–2232, 2015.

[10] Y. Jiang, H. Zhang, Z. Li, Y. Deng, X. Song, M. Gu, and J. Sun, "Design and optimization of multiclocked embedded systems using formal techniques," *IEEE transactions on industrial electronics*, vol. 62, no. 2, pp. 1270–1278, 2015.

[11] M. Zhou, F. He, B.-Y. Wang, M. Gu, and J. Sun, "Array theory of bounded elements and its applications," *Journal of automated reasoning*, vol. 52, no. 4, pp. 379–405, 2014.

[12] M. Zhou, F. He, X. Song, S. He, G. Chen, and M. Gu, "Estimating the volume of solution space for satisfiability modulo linear real arithmetic," *Theory of Computing Systems*, vol. 56, no. 2, pp. 347–371, 2015.

[13] Y. Jiang, H. Liu, H. Kong, R. Wang, M. Hosseini, J. Sun, and L. Sha, "Use runtime verification to improve the quality of medical care practice," in *2016 38th ACM International Conference on Software Engineering(ICSE)*, ACM, 2016.

[14] Y. Jiang, H. Song, R. Wang, M. Gu, J. Sun, and L. Sha, "Data-centered runtime verification of wireless medical cyber-physical system," *IEEE Transactions on Industrial Informatics*, 2016.

[15] C. J. Colbourn, S. S. Martirosyan, G. L. Mullen, D. Shasha, G. B. Sherwood, and J. L. Yucas, "Products of mixed covering arrays of strength two," *Journal of Combinatorial Designs*, vol. 14, no. 2, pp. 124–138, 2006.

[16] A. W. Williams and R. L. Probert, "A practical strategy for testing pairwise coverage of network interfaces," in *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pp. 246–254, IEEE, 1996.

[17] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE software*, vol. 13, no. 5, pp. 83–88, 1996.

[18] Y.-W. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings, 2000 IEEE*, vol. 1, pp. 431–437, IEEE, 2000.

[19] K. Lei and O. Kuhn, "Lawrence," ipog: A general strategy for t-way software testing,"," in *Proceedings IEEE Conference and Workshops on the Engineering of Computer-Based Systems*, 2007.

[20] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of strengths 2 through 6 using a tabu search approach," *Discrete Mathematics, Algorithms and Applications*, vol. 4, no. 03, p. 1250033, 2012.

[21] X. Liang, S. Guo, M. Huang, and X. Jiao, "Combinatorial test case suite generation based on differential evolution algorithm," *Journal of Software*, vol. 9, no. 6, pp. 1479–1484, 2014.

[22] M. B. Cohen, C. J. Colbourn, and A. C. Ling, "Augmenting simulated annealing to build interaction test suites," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pp. 394–405, IEEE, 2003.

[23] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pp. 365–373, IEEE, 2015.

[24] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 301–311, IEEE, 2013.

[25] K. Chandrasekar and N. Ramana, "Performance comparison of de, pso and ga approaches in transmission power loss minimization using facts devices," *Int J Comput Appl*, vol. 33, no. 5, pp. 58–62, 2011.

[26] A. Ouarda and M. Bouamar, "A comparison of evolutionary algorithms: Pso, de and ga for fuzzy c-partition," *International Journal of Computer Applications*, vol. 91, no. 10, 2014.

[27] P. Ouyang and V. Pano, "Comparative study of de, pso and ga for position domain pid controller tuning," *Algorithms*, vol. 8, no. 3, pp. 697–711, 2015.

[28] P. Rocca, G. Oliveri, and A. Massa, "Differential evolution as applied to electromagnetics," *Antennas and Propagation Magazine, IEEE*, vol. 53, no. 1, pp. 38–49, 2011.

[29] V. Nannen and A. Eiben, "Efficient relevance estimation and value calibration of evolutionary algorithm parameters," in *2007 IEEE Congress on Evolutionary Computation*, pp. 103–110, IEEE, 2007.

[30] V. Nannen and A. E. Eiben, "Relevance estimation and value calibration of evolutionary algorithm parameters.," in *IJCAI*, vol. 7, pp. 6–12, 2007.

[31] P. Balaprakash, M. Birattari, and T. Stützle, "Improvement strategies for the f-race algorithm: Sampling design and iterative refinement," in *International Workshop on Hybrid Metaheuristics*, pp. 108–122, Springer, 2007.

[32] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: an automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[33] J. Bach, "Allpairs." http://www.satisfice.com/, March 2016.

[34] D. Pallas, "jenny." http://burtleburtle.net/bob/math/jenny.html, March 2016.

[35] SigmaZone, "Pro-test software for efficient test case generation." http://www.sigmazone.com/protest.htm, March 2016.

[36] "Testcover." https://testcover.com/sub/tcgf.php, March 2016.

[37] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification*, pp. 504–518, Springer, 2007.

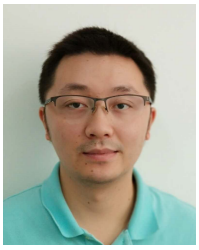
[38] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *2007 IEEE International Conference on Software Maintenance*, pp. 255–264, IEEE, 2007.

[39] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 540–550, IEEE Press, 2015.

[40] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 26–36, ACM, 2013.

[41] J. Petke, M. B. Cohen, M. Harman, and S. Yoo, "Practical combinatorial interaction testing: empirical findings on efficiency and early fault detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 901–924, 2015.

Yuexing Wang received the B.S. degree from the Department of Computer Science and Technology, Nankai University, in 2015. He is currently working toward the PhD degree at the School of Software, Tsinghua University. His research interests include software testing and program analysis.



Min Zhou is a research assist at the School of Software, Tsinghua University. He received his B.S. degree in the Department of Mathematical Science in 2007 and Ph.D. degree in the Department of Computer Science and Technology in 2014. His research interests include model checking, program analysis and testing.

Xiaoyu Song received the PhD degree from the University of Pisa, Italy, in 1991. From 1992 to 1998, he was on the faculty at the University of Montreal, Canada. He joined the Department of Electrical and Computer Engineering at Portland State University in 1998, where he is now a professor. He was an editor of IEEE Transactions on VLSI Systems and IEEE Transactions on Circuits and Systems. He was awarded an Intel Faculty Fellowship from 2000 to 2005. His research interests include formal methods, design automation, embedded systems and emerging technologies.

Ming Gu is a researcher, vice-director of School of Software, Tsinghua University, vice-director of Ministry of Education Key Laboratory of Information Security. Her research areas include software formal methods, software trustworthy, and middleware technology.

Jiaguang Sun is a professor, member of the Chinese Academy of Engineering, director of Ministry of Education Key Laboratory of Information Security, director of Tsinghua National Laboratory for Information Science & Technology (TNList), president of Executive Council of China Engineering Graphics Society. His research areas include computer graphics, computer-aided design, formal verification of software, software engineering, and system architecture.