A three-phase approach to improve the functionality of *t*-way strategy

Einollah Pira[1] Vahid Rafe[2,3] Sajad Esfandyari[2]

[1]Faculty of Information Technology and Computer Engineering, Azarbaijan Shahid Madani University, Tabriz 5375171379, Iran

[2]Department of Computer Engineering, Faculty of Engineering, Arak University, Arak 38156-8-8349, Iran

[3]Department of Computing, Goldsmiths University of London, London, UK

pira@azaruniv.ac.ir v.rafe@gold.ac.uk sajad.a1367@gmail.com

Abstract

Although *t*-way strategy tries to generate a minimum test suite (TS) for detecting errors in software systems, its functionality is affected by three important challenges. The first one, which relates to the quality of the generated TS, expresses that some complex errors (e.g. deadlocks in concurrent systems) may not be detected through the generated TS. The second one is that manually preparing parameters and their values in the modern software systems is difficult or even impossible, whereas the third one is the low generation speed and the large size of the generated test suite. In this paper, we propose a three-phase approach (so-called TPA) to handle these challenges. It seems that injecting some information about special errors into the test suite can raise its quality. For this purpose, TPA, in the first phase, uses an optimized version of model checking to extract such information from a model of the system under test. The extracted information is then injected into the test suite. In the second phase, TPA uses the generated state space in the first phase to automatically prepare parameters and their values. In the last phase, TPA applies an adopted version of evolution strategy to improve the functionality of *t*-way strategy in terms of generation speed and test suite size. Multiple and pairwise comparisons of results confirm that TPA has the best functionality in comparison with other evolutionary algorithms.

Keywords: Combinatorial testing, *T*-way strategy, Model checking, Evolution strategy, Interaction strength

1. Introduction

Testing is one of the most important steps in the software development life cycle to detect any gaps, errors, missing requirements, and others [1]. Testing should check the system under test (SUT) in two aspects of structural and functional. In the structural one (also called white-box testing), the internal structure, design and coding of SUT are examined. Conversely, in the functional one (also called black-box tesing), the internal structure of SUT is ignored and the inputs and expected outputs (of course, from the customer's point of view) are considered to validate the overall functionality of SUT [2]. To success the black-box testing, all possible combinations of input parameters should be checked and this may expose the combinatorial explosion problem in large and complex systems. To handle this problem, there are several black-box methods such as cause-effect graph (CEG) and combinatorial testing (CT). CEG is a specification-based testing method that extracts the parameters and their values from a set of specified requirements. Actually, it graphically demonstrates the relationship between conditions (causes) and actions (effects) by which a truth table of the causes and the effects can be derived [3]. Similar to CEG, CT is a specification-based strategy that uses a systematic way (namely, *t*-way strategy or *t*-way interaction) to

generate a minimized TS such that it covers the required combinations of input parameters in accordance with the strength of interaction (i.e. *t* in the *t*-way strategy).

Usually, the functionality of *t*-way strategy is affected by three important challenges. The first one says that some complex errors (e.g. deadlocks in concurrent systems) may not be found through the generated TS. Actually, the none of test cases in the TS may detect such errors. The second challenge is about how to set input parameters and their values of the CT, manually or automatically. In most of the existing methods, this information are usually set manually and this causes it is difficult or even impossible in the modern software systems. The third challenge relates to the low generation speed (so-called performance) and the large size (so-called efficiency) of the generated test suite.

In this paper, we propose a three-phase approach (so-called TPA) to handle these challenges (i.e. each phase for a challenge). In the first phase, TPA uses an optimized version of model checking (OMC) to extract the information of special errors from a model of SUT. OMC employs Bayesian Optimization Algorithm (BOA) to explore the state space intelligently. BOA is an Estimation of Distribution Algorithm (EDA) that learns a Bayesian network (as a probabilistic model) from the current population and samples the network to generate new solutions [4]. In this context, a solution (or chromosome) is defined by a path in the state space that has a specific length and starts from an initial state. Upon finding an error, the exploration is stopped. Otherwise, after exploring the pre-specified number of states, the last state of a chromosome with the highest fitness value is considered as a pseudo-error. Due to the detected error (or pseudo-error), the required combinations of input parameters and their values are inserted into the test suite.

To extract the parameters and their values from the model, multi-stage algorithms have been proposed in the literature [5]. These approaches, although useful, can only work in simple cases. Moreover, they cannot detect the values of dynamic parameters. To reslove these problems, a novel approach has been proposed to automatically extract input parameters and their values from a model of SUT using model checking [6]. In this approach, after generating the state space entirely, the mentioned information is extracted. Although the approach can address the second challenge somewhat, it cannot work successfully in a large model of SUT due to the state space explosion problem. Unlike this approach, TPA, in the second phase, prepares parameters and their values without the state space explosion problem.

To handle the third challenge, several methods using evolutionary algorithms such as Genetic Algorithm (GA) [6], [7], Particle Swarm Optimization (PSO) [8], Cuckoo Search (CS) [1], and Ant Colony Algorithm (ACA) [7] have been recently proposed. Although promising, there is still room for improvement. In the last phase, TPA applies an adopted version of evolution strategy (so-called AES) to improve the functionality of CT in terms of generation speed and test suite size. AES is repeated until a test suite with full coverage is generated successfully or its execution time exceeds 24 h. In each iteration, AES, begins with only one random chromosome (as a test case) instead of a population of chromosomes, and it tries to increase its fitness (the coverage amount). Finally, TPA adds the achieved chromosome (test case) into the test suite.

The implementation results of TPA in the GROOVE toolset show that this approach improves the functionality of CT in terms of efficiency and performance. GROOVE is an open source toolset for designing and model checking graph transformation systems (GTS) [9]. TPA can generate test suites with the interaction strength up to $t = 25$.

The rest of the paper is organized as follows: In Section 2, we describe the required background such as BOA, GTS, MC, and *t*-way strategy. Section 3 surveys the related work. Section 4 is dedicated to describe the details of the first and second phases of TPA. In section 5, we present the third phase of TPA with more details. In Section 6, we evaluate TPA through different benchmark experiments. Threats to validity are discussed in Section 7. Finally, we conclude the paper and present ideas for our future works in Section 8.

2. Background

2.1. Bayesian Optimization Algorithm

Bayesian Optimization Algorithm (BOA) is one type of EDA which has solved a group of complicated problems such as Software Testing [10], Protein Folding [11], Planning Problems [12], and Searching the Optimum Path in 3D Spaces [13]. BOA uses a Bayesian network (BN) to consider multivariate interactions between variables. In other words, BOA applies the learning and sampling of a BN instead of the traditional genetic operators like crossover and mutation [4]. BN is a probabilistic graphical model which represents a problem as a set of random variables and probabilistic dependencies among them [4]. Usually, a BN is illustrated by a Directed Acyclic Graph (DAG) composed of nodes and arcs, which represent respectively random variables and probabilistic dependencies between these variables. The amount of probabilistic dependencies for each node is specified by a table. The components of a BN can be specified either manually by a domain expert or automatically using machine learning algorithms.

Similar to the other evolutionary algorithms such as GA, BOA produces the initial population of candidate solutions randomly. BOA then repeats several operations until the termination criteria such as an optimum solution is found or a maximum number of iterations is performed: (1) the current population is evaluated by a fitness function. (2) A set of most promising solutions are chosen and a BN is learned through these solutions. (3) The learnt BN is used to produce new candidate solutions and they are replaced with the ones in the current population that have the lowest fitness values [14].

2.2. Graph Transformation System

Graph Transformation System (GTS) is a formal language based on the concept of graphs which uses graphs and graph transformations to describe states and behaviors of a system respectively [15]. To describe a model, GTS uses a tuple (*TG*, *HG*, *R*) in which *TG* is a type graph, *HG* is a host graph, and *R* is a set of transformation rules. All node and edge types of a system is represented by *TG*. Moreover, *TG* includes two functions to determine source/destination nodes of edges. *HG* represents the initial state of the system and each node/edge in *HG* should be an instance of a node/edge type in *TG*, i.e. these graphs should be isomorphic. In a graph transformation rule *p*: *LHS* → *RHS*, *LHS* (also called left-hand side) specifies pre-conditions of *p* as well as *RHS* (also called right-hand side) determines post-condition of *p*. In simple terms, *LHS/RHS* sholud be in the current host graph before/after applying

the rule. Similar to *HG*, *LHS* and *RHS* should be isomorphic with *TG*. Some rules may have a *NAC* (negative application condition), which specifies a condition that its existence in the host graph causes the rule cannot apply.

To design a model of a system using GTS, several toolsets such as AGG [16] and GROOVE [9] have been developed. Among these tools, GROOVE has many important features that cause this tool is selected as a framework to implement the proposed approach. One of these features is that GROOVE can perform automatic verification (model checking) by production of the model's state space. Another feature is that this toolset is open source and we can add new capabilities to it. As an example of a system modeled in this toolset, the dining philosophers problem (DPP) with two philosophers is considered. Fig. 1 displays the host graph of the designed model. As shown in this figure, the nodes of this graph are $\{n_0, n_1, n_2, n_3\}$ in which n_0 and n_1 have the same type (i.e. Phil) and also the type of n_2 and n_3 is the same (i.e. Fork). It should be noted that the host graph in Fig.1 represents the initial state of DPP with two philosophers in the *thinking* mode and two forks in the *free* mode. The current status of philosophers or forks, which is composed of some attributes along with their values, are written within the corresponding nodes in the forms of expression of (Attribute Name = "*Attribute Value*").

To represent a transformation rule, GROOVE merges the graphs of *LHS*, *RHS*, and *NAC* (if available) and uses coloured coding to recognize their original graphs. In this coding, the components of *LHS* that should be removed after rule application, are colored with blue, whereas the ones of *RHS*, created after rule application, are colored with green. The common components of *LHS* and *RHS* are also colored with black color. Moreover, the components of *NAC* are colored with red color. For more information and examples about this subject, interested readers can refer to [17].

To generate the state space of a system, all transformation rules should be applied on the initial state repeatedly. The states space is usually represented in the format of a graph that nodes and edges specify states and transitions (i.e. applied rules on the states) between them respectively [18]. For example, Fig. 2 displays the state space of the DPP's model with two philosophers. In this figure, *s0* and *s12* denote to the initial and final states respectively. In the state space, a state with no output transition is called a final (or deadlock) state. *s12* describes the state of DPP in which all philosophers have picked up their left fork and are waiting for their right fork. Sometimes, some complex behavior of a model is described by a path, an alternating sequence of states and the applied rules (transitions) on them. For example, the path "*s0 go_hungry s*2 *get_left s5 go_hungry s*8 *get_left s12*" in Fig. 2 displays how the behavior of philosophers causes a deadlock in the system. Fig. 3 also shows the corresponding graph of the deadlock state *s12*.

2.3. Model Checking

Model checking is a formal verification technique which verifies/refutes behavioral properties of a given system through systematic inspection of all states (state space) of an appropriate model of the system [19]. Hence, the first step in this technique is to design a suitable model of the system and describe properties by a temporal logic. Generating the state space of the model entirely is the second step. In the third step, the generated state space is explored to examine a given property in each state. If the property is verified (or refuted) in a state, a witness (or counterexample) will be produced. A witness or (counterexample) is a path in the state space that starts from the initial state and ends in the goal state, i.e. a state in which the property is satisfied (or violated) [20]. There are two

well-known temporal logics, namely LTL[1] and CTL[2]. Each formula in LTL describes an infinite sequences of states in which each state at any point in time has only a unique successor, whereas in CTL, each state allows to have several successors. In CTL, the formula to describe the properties is linear in both the size of the state space and the size of the formula. Hence, in this paper, we have selected CTL to describe the properties.

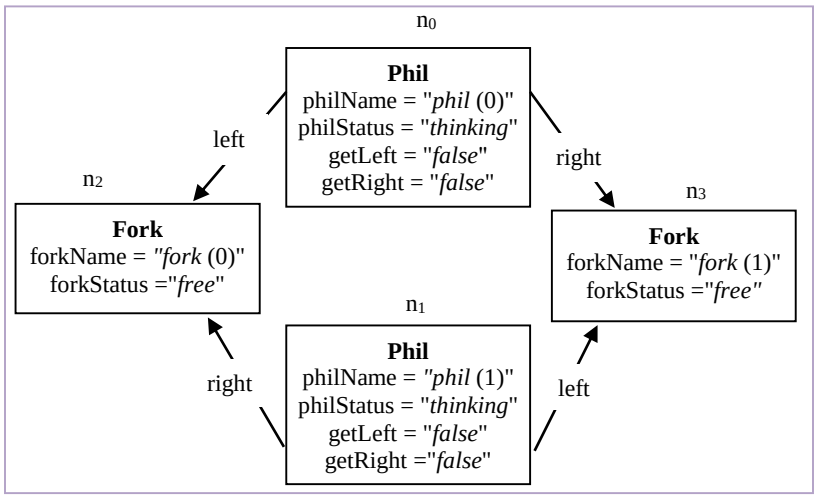

Fig. 1. Some details of DPP's model with two philosophers designed in GROOVE

Safety is one of the important properties of systems which can be checked by model checking. A safety property asserts that a bad state never occurs in the state space. Suppose that *q* is a bad state (e.g. *deadlock*), the property "*there isn't any q state in the given system*" can be a safety property provided that there shouldn't any *q* state in the state space. The formula *AG !q* is used in CTL (and also in GROOVE) to describe this safety property. In CTL, symbols *A* and *G* denote respectively to all possible paths in the state space and all states in each of these paths.

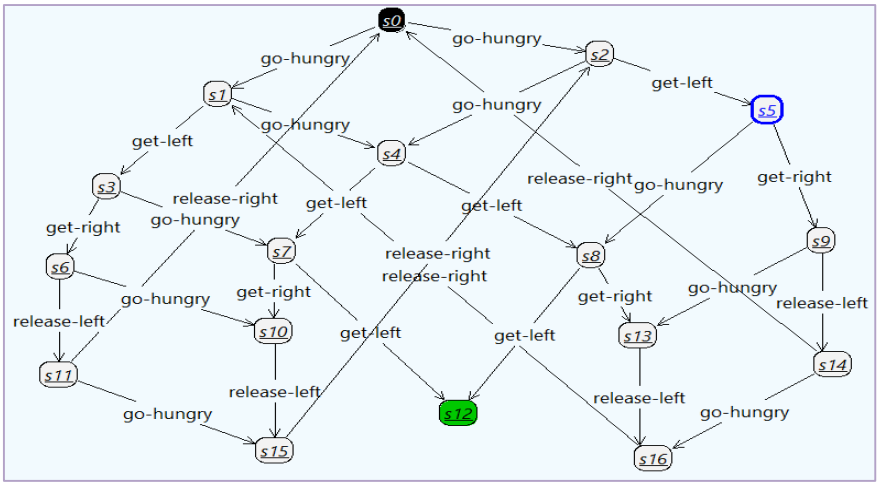

Fig. 2. The state space of the DPP's model with two philosophers

[1] Linear Temporal Logic
[2] Computation Tree Logic

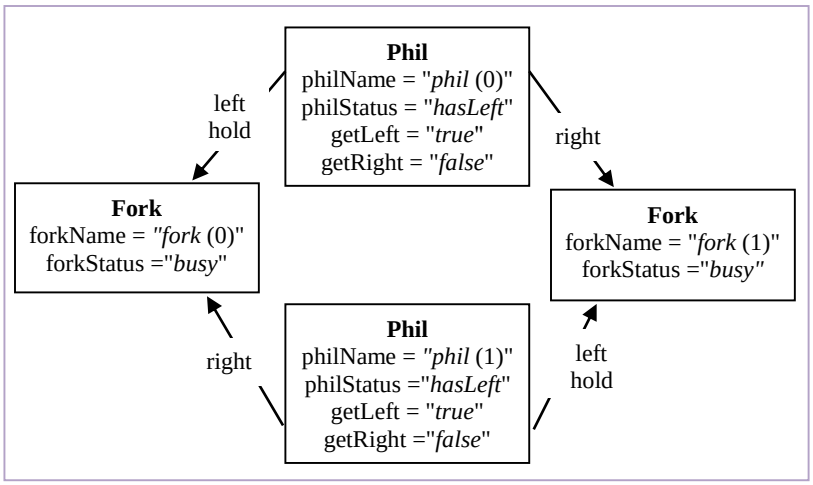

Fig. 3. The corresponding graph of the deadlock state of *s12*, shown in Fig. 2

2.4. *T*-way Strategy

Generating and checking all test cases in a real-world SUT may not be practical due to the large number of input parameters and their values. Assume that the set of $\{x_1, \ldots, x_p\}$ is input parameters of a dummy SUT such that each parameter x_i ($1 \leq i \leq p$) can get d_i different values. Each test case is shown by a p-tuple $(v_1, \ldots, v_p)$ in which v_i is one of d_i values, $1 \leq i \leq p$. For example, in DPP, each component is considered as an independent parameter and can get one or more values. Hence, in this system, the set of parameters will be {philName for the philosopher's name, philStatus for the philosopher's status, getLeft/getRight for the getting status of the left/right fork, forkName for the fork's name, forkStatus for the fork's status}. These parameters along with their values for a DPP's model with two philosophers are shown in Table 1 [6]. In this example, we have $p = 6$ and $d_1 = 2$, $d_2 = 5$, $d_3 = 2$, $d_4 = 2$, $d_5 = 2$, $d_6 = 2$. To test a SUT completely, entire possible settings of parameters should be generated. Each such setting is called a test case, and a set of test cases is also called a test suite. Obviously, a test suite should include all possible test cases in order to detect all existing errors. In the aforementioned example, the complete test suite contains 160 test cases (2 * 5 * 2 * 2 * 2 * 2 = 160). This number can be very large in complex SUT with high number of parameters, and this may expose the combinatorial explosion problem.

Table 1: All parameters along with their values for a DPP's model with two philosophers [6]

Parameter Name	philName	philStatus	getLeft	getRight	forkName	forkStatus
Parameter Values	*"phil* (0)" *"phil* (1)"	"*thinking*" "*hungry*" "*hasLeft*" "*eating*" "*hasRight*"	"*false*" "*true*"	"*false*" "*true*"	*"fork* (0)" *"fork* (1)"	*"free"* *"busy"*

A mathematical method to handle the combinatorial explosion problem is *t*-way testing in which *t* denotes the strength of interaction, and its value can be in the range of 2 to *p* [7]. In the *t*-way testing, the achieved test suite should contain only all *t*-way interactions instead of all *p*-way ones. Very small values for *t* cause that the required interactions to detect errors cannot be identified. Conversely, if *t* takes large values, the combinatorial explosion

problem occurs again. Therefore, it is important to find a suitable value for t. The generated test suite by t-way testing is called Covering Array (CA). In the dummy SUT, mentioned in above, the covering array is denoted by CA (N; t, p, d_1, …, d_p) where N is the number of test cases and t is covering strength. Of course, if all d_i ($1 \leq i \leq p$) are equal with d, it is denoted by CA (N; t, p, d) or briefly CA (N; t, d^p). Formally, CA is a two-dimensional array $N \times p$ with two following properties [21]. (1) For all i in the range of 1 to p, each of d_i values of parameter x_i should be seen in column i of the array at least once. (2) All combinations of each t parameters should be covered by a sub-array $N \times t$.

As mentioned before, the complete test suite in the DPP's model with two philosophers contains 160 test cases. By applying 2-way testing, the generated test suite, i.e. CA (N; 2, 6, $2^5 5^1$), will have 10 rows (N = 10). Fig. 4 displays this test suite along with the complete one. As displayed in the Table B of this figure, all values of each parameter (shown in Table 1) are seen in the corresponding column of this table at least once. For example, all values of philStatus, which are {"*thinking*", "*hungry*", "*hasLeft*", "*eating*", "*hasRight*"}, are seen in column 2 of Table B. Moreover, all combinations of each two parameters are covered by a sub-array 10 × 2 of Table B. For example, two parameters philName and philStatus have 2×5 = 10 different combinations which have covered by columns 1 and 2 of Table B.

A: The complete test suite

Test Case Number	philName	philStatus	getLeft	getRight	forkName	forkStatus
1	"*phil* (0)"	"*thinking*"	"*false*"	"*false*"	"*fork* (0)"	"*free*"
2	"*phil* (1)"	"*thinking*"	"*false*"	"*false*"	"*fork* (0)"	"*free*"
3	"*phil* (0)"	"*hungry*"	"*false*"	"*false*"	"*fork* (0)"	"*free*"
4	"*phil* (1)"	"*hungry*"	"*false*"	"*false*"	"*fork* (0)"	"*free*"
. . .	. . .	. . .	. . .	. . .	. . .	. . .
159	"*phil* (0)"	"*hasRight*"	"*true*"	"*true*"	"*fork* (1)"	"*busy*"
160	"*phil* (1)"	" *hasRight* "	"*true*"	"*true*"	"*fork* (1)"	"*busy*"

Applying 2-way testing

B: The generated test suite, i.e. CA (10; 2, 6, $2^5 5^1$)

Test Case Number	philName	philStatus	getLeft	getRight	forkName	forkStatus
1	"*phil* (0)"	"*hungry*"	"*true*"	"*false*"	"*fork* (1)"	"*free*"
2	"*phil* (1)"	"*hasRight*"	"*false*"	"*false*"	"*fork* (0)"	"*busy*"
3	"*phil* (0)"	"*thinking*"	"*false*"	"*true*"	"*fork* (0)"	"*free*"
4	"*phil* (1)"	"*hasLeft*"	"*true*"	"*true*"	"*fork* (1)"	"*busy*"
5	"*phil* (0)"	"*eating*"	"*false*"	"*true*"	"*fork* (1)"	"*busy*"
6	"*phil* (1)"	"*eating*"	"*true*"	"*false*"	"*fork* (0)"	"*free*"
7	"*phil* (0)"	"*hasRight*"	"*true*"	"*true*"	"*fork* (1)"	"*free*"
8	"*phil* (1)"	" *hungry*"	"*false*"	"*true*"	"*fork* (0)"	"*busy*"
9	"*phil* (1)"	"*thinking*"	"*true*"	"*false*"	"*fork* (1)"	"*busy*"
10	"*phil* (0)"	"*hasLeft*"	"*false*"	"*false*"	"*fork* (0)"	"*free*"

Fig. 4. The complete test suite and generated test suite for CA (N; 2, 6, $2^5 5^1$)

3. Related Work

The state-of-the-art researches to improve the performance and the efficiency of *t*-way strategy can be grouped in four categories. The first category includes random methods that choose test cases randomly using input distribution. The most application of this method is to compare the effectiveness of it and other generation algorithms [22]. The second category contains mathematic methods which use the concept of mathematical functions such as Orthogonal Array (OA) to generate optimal CA [1]. In [23], the authors apply a recursive technique to build larger CAs from smaller CAs. Moreover, several tools are designed based on the extension of the OA construction, e.g., Combinatorial Test Services (CTS) [25] and Test Configuration (Tconfig) [26] strategies. Although these methods are based on the lightweight computations, they can generate optimal CAs only for small and special configurations.

The mehtods in the third category use greedy algorithms to produce test cases that cover all possible combinations according to the input specifications. These algorithms use two different approaches: One-Parameter-at-a-Time (OPT) and One-Row-at-a-Time (ORT) [27]. In OPT, the CA is firstly constructed for two parameters by adding row by row. Afterwards, the CA is reconstructed by inserting more parameters, and this process continues until all parameters are considered. The first implementation of this method is In-Parameter-Order (IPO) algorithm [28]. Based on IPO, other strategies like In-Parameter-Order-General (IPOG) [29] and In-Parameter-Order-General-Fast (IPOG-F) [30] are produced. As the name of ORT implies the CA is built row by row. The inserted row should cover all *t*-tuples as much as possible. Upon covering all *t*-tuples, ORT will stop [1]. The Automatic Efficient Test Generator (AETG) is the first ORT-based strategy to generate the test suite. It chooses one test case among several candidate test cases in a greedy fashion, and adds it to the test suite [31]. In addition to AETG, other ORT-based strategies such as Pairwise Independent Combinatorial Testing (PICT) [32] and GTWay [33] can be mentioned.

The fourth category includes methods which use meta-heuristic algorithms. These methods, which use the ORT-based strategy, generate a random set of test cases, and apply a series of operators (e.g. mutation and crossover ones in GA) on the test cases to improve them. Afterwards, one test case with the highest fitness is selected and added to the test suite. This process will be continued until all interactions are covered by the test suite. In [34], the authors used SA, GA and TS algorithms for solving 2-way interaction. Experimental results show that SA outperforms GA and TS. In [35], the SA algorithm is extended to support VSCA up to t = 3. Moreover, in [36], the authors proposed a strategy based on PSO algorithm to generate the test suite. This strategy, which is called Fuzzy Self Adaptive PSO (FSAPSO), employs fuzzy techniques to compute the PSO parameters. Experimental results show that FSAPSO can produce test suites up to t = 4, and also it outperforms GA in most of configurationsthe. As other strategies based on PSO, FSAPSO [37], Particle Swarm-based t-way Test Generator (PSTG) [38], Hybridized Bat Algorithm and Particle Swarm Optimization (BAPSO) [39], and Discrete Particle Swarm Optimization (DPSO) [40] can be mentioned. In [41], an approach based on harmony search algorithm is proposed. This approach, which is called HSS, simulates the behavior of musicians to make the best song. HSS can support much higher strengths (up to t = 15). In [42], an approach based on Teaching-Learning-Based Optimization (TLBO) is proposed. This approach, which is called ATLBO, applied hash map for faster access to test cases. Experimental results show that ATLBO is better than TLBO in terms of efficiency and performance.

As other methods based on GA, PwiesGen [43], and Pairwise Test Sets (GAPTS) [44] strategies can be mentioned. These strategies usually reach to the optimum test suites after a lot of iterations, as a result, they have the low generation speed. To handle this drawback, the strategy presented in [45] applies some changes in the structure of chromosomes. Experimental results show that this strategy is better than PwiesGen, GAPTS, and PWiesGenPM in terms of the test generation time. One of the powerful algorithms in the context of CT is GS [21]. GS, which is based on GA, is improved in terms of efficiency and performance by changing the bit structure, quick accessing to test cases, and adjusting the crossover and mutation operators. Moreover, GS can support much higher strengths (up to t = 20). In [6], the authors proposed an approach to extract input parameters and their values from a model of SUT using model checking automatically. Like to GS, this approach can support much higher strengths (up to t = 20).

In [46], the authors proposed the Gravitational Search Test Generative (GSTG) strategy originated from the gravitational interaction between objects. In GSTG, test cases and their weight are specified by objects and their mass respectively. Due to absorbing lighter objects by heavier ones with the force of gravity, test cases with less weight move slowly towards test cases with more weight. As a result, a test case with the highest weight will be generated. The experimental results show that GSTG can support much higher strengths (up to t = 16). Multiple Black Hole (MBH) [47] is another meta-heuristic strategy based on the behavior of the black hole and its around stars. An advantage of MBH is that it doesn't need to tune multiple parameters. The evaluation results show that MBH can produce a covering array up to t = 4. Sine Cosine Variable Strength (SCAVS) [48] is another strategy which uses the mathematical characteristics of the sine and cosine trigonometric functions. SCAVS can generate a covering array up to t = 6. Artificial Bee Colony (ABC) algorithm [49] simulates the group behavior of honey bees in the nature. Based on this algorithm, several strategies such as Artificial Bee Colony (ABC) [50] and Artificial Bee Colony Variable Strength (ABCVS) [51], and Hybrid Artificial Bee Colony (HABC) strategy [52] are presented to generate the minimum covering array.

A hybrid strategy based on DPSO and GS (GALP) [53], Elitist Hybrid of the Migrating Birds Optimization algorithm and Genetic Algorithm (EMBO-GA) [54], African Buffalo Optimization (ABO) [55], Auto Constrained Covering Array Generation (AutoCCAG) [56], Success-History and Linear Population Size Reduction based Adaptive Differential Evolution strategy (LSHADE) [57], and Bi-objective Dragonfly Algorithm (BDA) [58] are examples of other meta-heuristic strategies to generate the minimum covering array.

4. The first and second phases of TPA: Optimized model checking using BOA

As mentioned before, one of the important challenges affecting on the functionality of *t*-way strategy is that the generated test suite may not reveal some complex errors. For example, according to Fig. 3, two test cases of ("*phil* (0)", "*hasLeft*", *true*, *false*, *"fork* (0)", "*busy*") and ("*phil* (1)", "*hasLeft*", *true*, *false*, *"fork* (1)", "*busy*") should be in each generated test suite. Whereas, the test suite in Table B of Fig. 4, generated by GA [6], doesn't contain these test cases. To resolve this challenge, TPA uses an optimized version of model checking (OMC) that extracts the information about special errors from a model of a given SUT, and adds this information as test cases into the generated test suite. Traditional model checking usually explores all reachable states (i.e. the state space) of a given model thoroughly to check properties [59]. To prevent the state space explosion in very large models, OMC has been proposed [60]. OMC is a hybrid approach of model checking and BOA that tries to find an error (e.g.

deadlock) without exploring the entire state space. BOA applies a Bayesian network (BN) to capture the dependencies between problem variables from the current population and samples the network to generate new solutions [4]. In the BOA-based approach, proposed in [60] for detecting deadlocks in systems specified formally through GTS, the structure of BNs are supposed fixed and three different versions of the approach are proposed. The first version, which is called nBOA (naïve BOA), uses the naïve BN. Whereas, the second version, which is called cBOA (chain BOA), employes BNs with the structure of a chain in which each node depends conditionally on the previous node. Finally, the third version, which is called tpBOA (two parents BOA), uses the BNs with the structure of a chain such that each node depends conditionally on two previous nodes. According to the reported results, nBOA has a better performance in comparison with others. Therefore, OMC uses nBOA to find a deadlock state. If nBOA can find such state successfully, equivalent test cases are inserted into the test suite. In case of failure, the last state of a chromosome with the highest fitness value is considered as a pseudo-error and equivalent test cases are added into the test suite. More details about nBOA are given in [60].

Another important challenge affecting on the functionality of *t*-way strategy is that manually preparing parameters and their values in the modern software systems is difficult or even impossible. Although, a novel approach using model checking has been proposed to extract parameters and their values from a model of SUT automatically [6], it cannot work successfully in a large model of SUT due to the state space explosion problem. Unlike this approach, TPA, in the second phase, uses the information of explored states by OMC to prepare parameters and their values without any problem because OMC stops after exploring the pre-specified number of states. TPA checks each explored state and adds new parameters along with their values. For example, we consider the state space of the DPP's model with two philosophers (i.e. Fig. 2). The state *s0* represents the model's initial state and TPA can extract the initial values of parameters, i.e. "*phil* (0)" and "*phil* (1)" for philName, "*thinking*" for philStatus, "*false*" for getLeft, "*false*" for getRight, "*fork* (0)" and "*fork* (1)" for forkName, and "*free*" for forkStatus. Similar to *s0*, all other states are examined to find new parameters along with their values. Table 1 shows the extracted parameters along with their values for a DPP's model with two philosophers.

5. The third phase of TPA: an adopted version of evolution strategy

Evolution Strategies (ESs) which are based on ideas of evolution, use selection and mutation operators in order to iteratively evolve a population of individuals. A $(\mu + \lambda)$-ES is a general evolution strategy that selects μ number of most promising individuals from among the set of μ current parents and λ mutants (i.e. the results of parents' mutation). As mentioned before, the low generation speed and the large size of the generated test suite is considered as the third challenge of *t*-way strategy. To resolve this challenge, TPA, in the third phase, applies an adopted version of evolution strategy (so-called AES) which is a $(1 + 1)$-ES. It means that AES operates on only one random chromosome (as a test case) instead of a population of chromosomes. If the mutant is more promising than the parent one, it is considered as the parent of the next generation. Otherwise, the mutant is ignored. TPA firstly adds the test cases generated by OMC into an empty test suite, and then it repeats AES until the test suite covers all *t*-interactions successfully or its execution time exceeds 24 h. In each iteration, AES generates one chromosome (as a test case) randomly, and it tries to increase its fitness (weight or the coverage amount) during the different

generations. For this purpose, AES mutates the current chromosome by using q-mutation operator for all q in the range of 1 to p (the number of input parameters of SUT), and repeats this process for a predefined number of steps (namely, *maxGenerations*). Finally, TPA adds the achieved chromosome (test case) into the test suite. Fig. 5 shows the flowchart of the third phase of TPA. In Section 5.1, a fitness function is proposed to compute the weight of test cases. The q-mutation operator is also presented in Section 5.2.

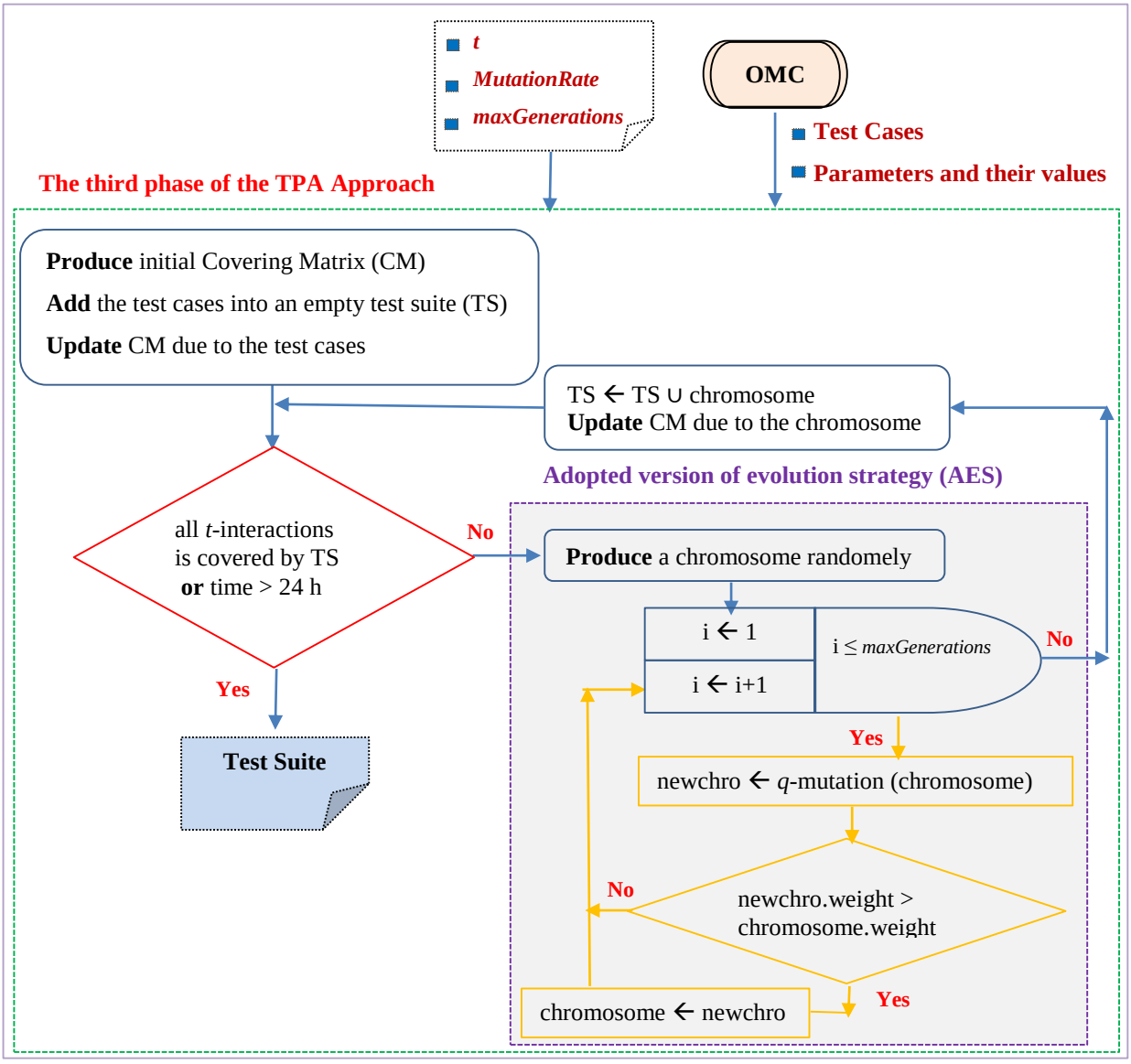

Fig. 5. The flowchart of the third phase of TPA

5.1. Fitness function

To calculate the weight of a test case, we firstly introduce the concept of covering matrix. Covering matrix (CM) is a data structure to calculate the coverage amount (weight) of a test case. CM should hold all t-interactions of p parameters. It means that, CM has $\binom{p}{t}$ rows, the first t columns (cells) of each row for combinations of t different parameters, and the rest cells except one for showing the covering status of these combinations. Also, the number of

the combinations is displayed in the last cell, and the summation of these numbers is called *goalCoverage*, i.e. the total number of combinations that should be covered by the test suite. For simplicity in calculations, numeric values are used instead of parameter names and values. It should be noted that the number of columns in rows can be different and they depend on the number of values for each parameter. For example, row 1 of the CM in Fig. 6. has 9 coulmns (2+2*3+1), whereas row 5 has 15 columns (2+3*4+1). At the beginning, which the test suite is empty, all cells for covering status are initialized with 0.

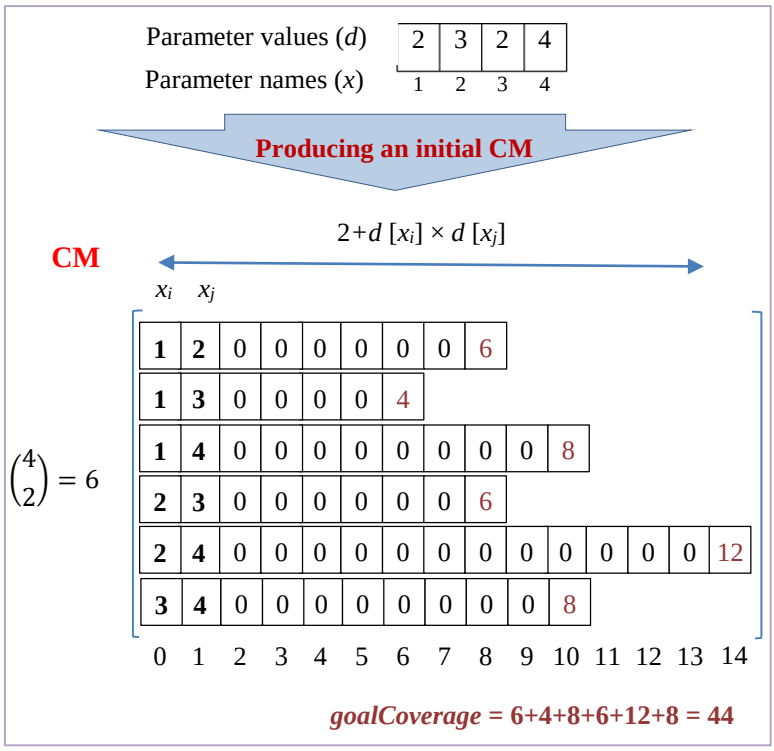

Fig. 6. An example of producing an initial CM

To produce a CM practically, the authors in [6] use nested loops with many loop variables due to the values of *t*. As a sample, for $t = 10$, nested ten-loops with ten loop variables are used. Actually, for each *r* in the range of 2 to *t* (the maximum value of *t* in [6] is 20), they use nested *r*-loops with *r* loop variables. Low readability of such coding is a drawback. As another drawback for such coding is that for large values of *t* ($t > 20$), the authors should modify the implementation. To resolve these drawbacks, in this paper, we use an array to store the current values of loop variables, and nested two-loops to simulate nested *r*-loops with *r* loop variables (for simplicity, *iloop* and *oloop* denote for the inner and outer loops respectively) [61]. The *iloop* is used to increment the innermost variable and to check if it spills over; in case of spilling over, it moves to upper level of the loop. The final condition, i.e. all *r* loop variables are reached to the upper bound, to stop the execution of loops is checked in *oloop*. Algorithm 1 shows the pseudocode of producing a CM with nested two-loops.

When we want to add a test case into the test suite, firstly its weight should be computed. If the weight isn't zero, i.e. it covers some combinations, it is added into the test suite. Of course, the CM should also be updated. To calculate the weight of a given test case, for each row of CM, due to *t* first columns and corresponding values in the

test case, the equivalent decimal value is computed (according to lines 5-13 of Algorithm 2). If the corresponding column of this value is zero, i.e. this combination isn't covered by the test case, the weight is incremented by one unit. Moreover, the corresponding column is set to one in order to show that this combination is covered by a test case (according to lines 14-17 of Algorithm 2). Otherwise, it means that this combination has been covered by previous test cases and so the weight will not be changed. Finally, the value of *goalCoverage* is updated. Fig. 7 illustrates an example of computing the weight of a test case and updating the CM.

Algorithm 1 Producing a CM with nested two-loops

```
1:  Input: p: the number of parameters, d: parameter values, t: the interaction strength;
2:  Output: CM: a covering matrix,  goalCoverage;
3:  ArrayList <int[]> CM = new ArrayList<int[]> () ;
4:  boolean status = false; int goalCoverage = 0;
5:  int totalUP = 0;  // the summation of all upper bounds of loop variables
6:  int[] UP = new int [t];  // for holding the upper bounds of loop variables
7:  for i = 1 to t do
8:        UP [i] = p - t + i;  totalUP = totalUP + UP [i];
9:  end for
10:  int[] arrs = new int [t];  // for holding the current value of loop variables
11: arrs [1] = 1;
12: for i = 2 to t do
13:        arrs [i] = arrs [i-1] +1;
14: end for
15: while not status do     // the outer loop, i.e. oloop
16:  if the summation of values in arrs is equal with totalUP then status = true;
17:  if all values in arrs are different then
18:      int len = the production of d [arrs [r]] for r = 1 to t;
19:      int[] a = new int[len + t +1];
20:     for r = 1 to t do
21:          a [r] = arrs [r];
22:      end for
23:      a [len + t + 1] = len; goalCoverage + = len; CM . add (a);
24:  end if
25:  boolean change = true; int r = t;  // start from the innermost loop
26:  while change and r > = 1 do   // the inner loop, i.e. iloop
27:      arrs [r] ++;    // increment the innermost variable and check if spill overs
28:      if arrs [r] > UP [r] then
29:                Reintialize loop variables; change = true;
30:      else
31:                change = false;
32:      end if
33:      r --;     //move to upper level of the loop
34:  end while
35: end while
```

5.2. *Q*-mutation operator

In this paper, *q*-mutation is the only evolution operator to improve the quality of generations. This operator selects *q* genes from the chromosome randomly and mutates all of them with a pre-defined probability (namely, *MutationRate*). To mutate a gene *i* ($1 \leq i \leq p$) of a test case $tc = (v_1, \ldots, v_p)$, v_i is replaced with a new value v'_i that is chosen among candidate values for this gene randomely. At first, the repetition number of all possible values of parameter *i* is calculated in the current test suite. The values with minimum repetition numbers are then considered as candidate values for this gene. For example, Fig. 8 displays the process of the 2-mutation operator on $tc = (1, 1, 1, 2)$. For simplicity, it is supposed that the possibility values for a parameter with *w* different values are a set of {0,1,…, *w*-1}. For example, the second parameter in this figure has 3 different values, and so the possibility values

for this parameter are a set of {0,1,2}. Accodring to this figure, two genes of 2 and 4 are selected randomly. Due to the current test suite, candidate values for each gene are computed and one of them is selected randomly. Finally, the achieved test case will be *tc* = (1, 2, 1, 1) that, as expected, it has the higher weight compared to the primary test case.

Algorithm 2 Computing the weight of a test case and updating the CM

```
1:  Input: ts: a test case, d: parameter values, t: the interaction strength, CM: the covering matrix;
2:  Output: CM: the covering matrix, weight, goalCoverage;
3:  int weight = 0;
4:  for i = 1 to CM.size() do
5:      int[] row = CM.get (i); int[]  pow = new int [t];
6:      pow [1] = 1;
7:      for j = 2 to t do
8:          pow [j] = d [row [j-1]] * pow [j-1];
9:      end for
10:     int c = t;
11:     for j = t downto 1 do
12:         c += ts [row [j]] * pow [j];
13:     end for
14:     if row [c] == 0 then
15:         row [c] = 1; row [row. size()] -=1; CM.set (i, row);
16:         weight ++;
17:     end if
18: end for
19: goalCoverage -= weight;
```

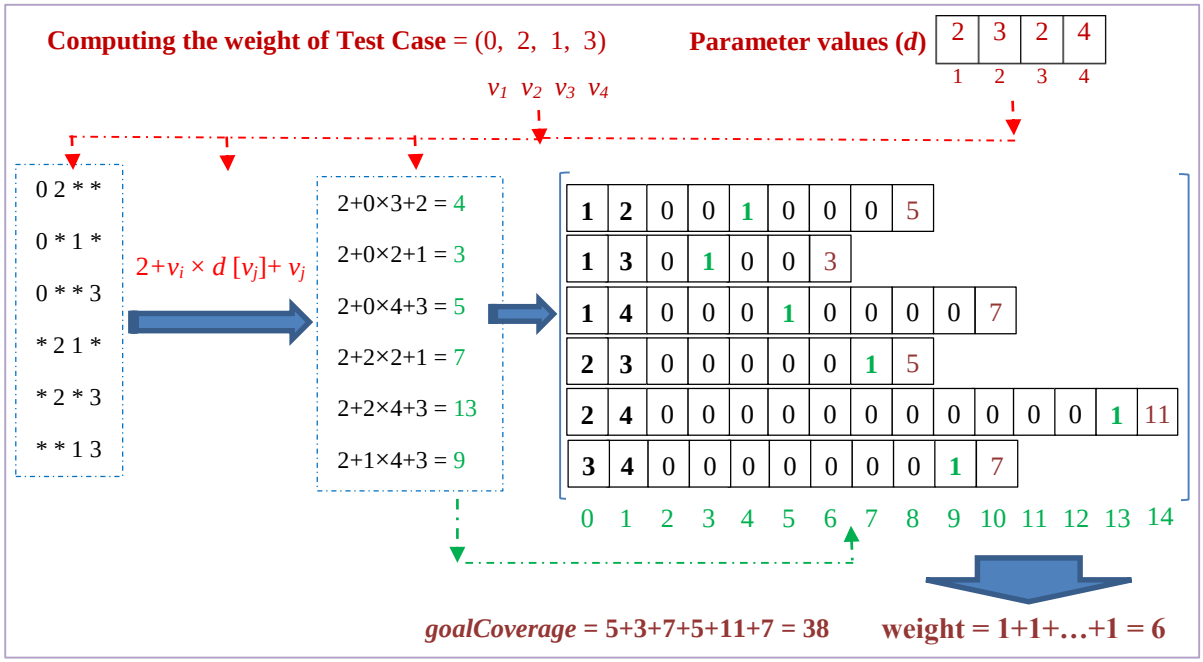

Fig. 7. An example of computing the weight of a test case and updating the CM

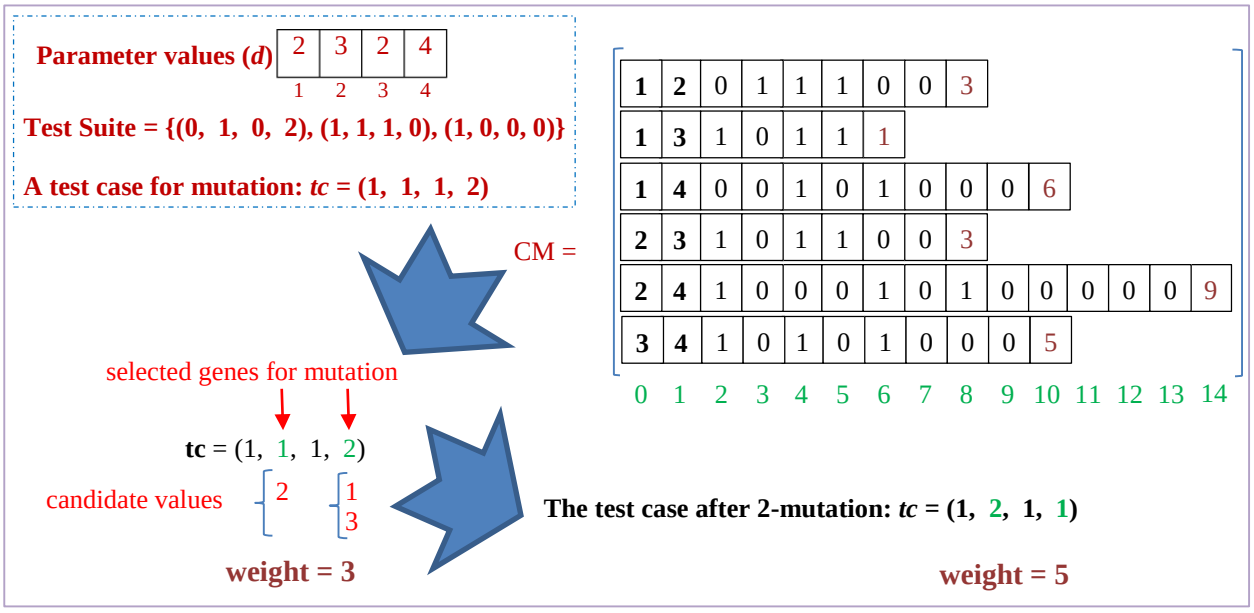

Fig. 8. The process of the 2-mutation operator

6. Evaluation results

To evaluate the efficiency and performance of TPA, the Java language is used to implement it in the GROOVE toolset. The results of TPA are compared with others in two ways that parameters and their values are prepared. In the automatically way, we use evolutionary approaches such as GS [21], TLBO [62], BA [63], and PSO [8], and to have a fair comparison, all approaches have been implemented in GROOVE. In the manually way, in addition to TPA and GS, we use TConfig as a mathematic method, PICT and IPOG as greedy algorithms, CS, PSTG, and DPSO as evolutionary algorithms. In result tables, the average test suite sizes and generation times over 20 independent runs of all approaches are displayed. It should be noted that all results have been obtained on a system with an Intel CORE i5 processor and 6 GB of memory. In the result tables, if an approach doesn't support corresponding configuration, the term "NS" is used. Moreover, the term ">day" is used when the approach cannot generate a complete test suite in less than a day. For more readability, the best results in each row are highlighted.

Before executing the approaches, suitable values for the parameters should be specified. The values for the parameters of previous approaches have been taken from related articles. Fortunately, TPA has only two important parameters: *MutationRate* and *maxGenerations*. To find a proper value for *MutationRate*, we have executed TPA with different values for this parameter. The value by which TPA has the best results is 0.8, which it is relatively large. Of course, it is expected because TPA is based on mutations only, and to increase the weight of a chromosme, mutations should be performed in most cases. Moreover, after executing TPA with different values for *maxGenerations*, the value of 100 is considered as the best value. Table 2 shows these parameters along with their proper values.

Table 2: Proper values for parameters of the used approaches

Approach	Parameter	Value
TPA	*maxGenerations*	100
	MutationRate	0.8
GS [21]	*Population*	50
	Crossover rate	0.4
	Mutation rate	0.6
	Selection method	Tournament selection
PSO [8]	*Population*	80
	W	0.8
	C1	2
	C2	3
BA [63]	*Population*	80
	Min frequency	0
	Max frequency	100
	Loudness	0.25
	Pulse rate	0.5
TLBO [62]	*Population*	80

In the automatically way, we employ different models of five known case studies, namely Dining Philosophers Problem (DPP) [64], Process Life Cycle Problem (PLC) [65], Online Shopping System (OSS) [60], Firewalls System (FWS) [66], and Hotel Booking System (HBS) [67]. It should be noted that considered deadlock states in some case studies may not show the actual errors, but since these states don't have any output transitions in the state space, they can be considered as deadlocks in order to evaluate TPA and the others. In the following, we give a brief description of these case studies along with running results of all approaches on their sample models.

DPP, which is used to check the correctness of concurrent algorithms, describes the behaviours of several philosophers seating around a table, and a fork is placed between each neighboring pair of philosophers. The initial mode of all philosophers is *thinking*. After a while, each of these philosophers can modify its mode to *hungry* and he/she can then pick up his/her left fork provided that it is free. In the case of obtaining the left fork, his/her mode is changed into *hasLeft*. If a right fork of a philosopher with *hasLeft* mode is free, he/she picks up it and modifies his/her mode to *eating*. Finally, an *eating* philosopher releases the obtained forks and goes to *thinking*. If all philosophers pick up their left forks, it is called that DPP has faced with a deadlock state, i.e. an error.

Table 3 displays average test suite sizes and generation times for three different models of DPP. This case study has six parameters (i.e. $p = 6$) and interaction strength (i.e. t) is between 2 and 5. As seen in this table, TPA outperforms the others in terms of average generation time (i.e. performance) in all cases. At $t = 2$, all approaches generate test suites with equal sizes (i.e. efficiency). Whereas, in some configurations (three ones), GS and PSO have the high efficiency. In rest configurations, TPA outperforms the others in terms of efficiency.

PLC is used to describe the life cycle of processes in the operating system. Each newly created process is loaded into the memory if the memory is available. The loaded process can request devices such as CPU or I/O, that if they are free, the process gets them and executs, otherwise, it waits. After completing, the process frees all allocated devices and stops. In this problem, a deadlock state occurs when all processes have executed and stopped. Table 4 displays the parameters name along with their values for a sample model of this case study with 5 processes

and 3 memory segments. As seen, this model has 9 parameters with different values of $d_1 = 5$, $d_2 = 8$, $d_3 = 3$, $d_4 = 2$, $d_5 = 2$, $d_6 = 2$, $d_7 = 2$, $d_8 = 2$, and $d_9 = 2$. Therefore, the covering array for this model will be CA (N; t, 9, $5^1 8^1 3^1 2^6$).

Table 3: Average test suite sizes and generation times for DPP (p = 6), with t varied up to 5.

t	the size of model	TPA Size/Time (s)	GS Size/Time(s)	PSO Size/Time(s)	TLBO Size/Time(s)	BA Size/Time(s)
2	10 phils	100.98/0.86	100.56/1.29	100.49/1.19	100.69/1.84	100.78/1.25
	11 phils	121.52/0.78	121.63/1.51	121.76/1.56	121.51/2.11	121.53/1.43
	12 phils	144.58/0.79	144.52/1.76	144.87/1.74	144.92/2.16	144.98/2.14
3	10 phils	503.93/0.93	505.83/6.14	506.31/6.54	504.36/9.31	507.03/6.87
	11 phils	608.21/0.95	610.53/7.68	612.42/7.96	611.5/11.36	610.53/7.58
	12 phils	722.54/0.99	724.62/9.59	725.63/9.21	723.51/12.69	726.5/9.41
4	10 phils	1016.79/1.12	1015.31/11.73	1013.23/11.22	1018.43/16.98	1015.83/11.39
	11 phils	1226.84/1.21	1221.72/13.89	1224.31/16.09	1235.31/21.05	1226.39/14.42
	12 phils	1452.52/1.32	1449.16/16.05	1450.31/16.69	1465.30/23.73	1451.52/15.81
5	10 phils	2162.4/1.65	2161.8/15.09	2173.6/16.08	2168.66/21.36	2167.33/16.17
	11 phils	2614.85/2.31	2613.4/17.42	2626.33/19.54	2645.32/30.28	2629.63/20.26
	12 phils	3085.31/3.14	3097.34/21.94	3098.32/22.78	3096.42/33.42	3099.25/23.51

In Table 5, average test suite sizes and generation times are illustrated for two different models of PLC. This case study has 9 parameters (i.e. p = 9) and interaction strength (i.e. t) is between 2 and 8. According to this table, TPA has the best performance in all configurations. Although, TPA has the lower (or equal) efficiency compared to the others in some configurations, it outperforms them in most ones.

Table 4: All parameters along with their values for a PLC's model with 5 processes and 3 memory segments

Parameter Name	Process Name	Process Status	Memory Name	Memory Status	CPU Status	IO Status	Get CPU	Get IO	Get Memory
Parameter Values	*"P (0)"* *"P (1)"* *"P (2)"* *"P (3)"* *"P (4)"*	*"idle"* *"stopping"* *"running"* *"exeIO"* *"waiting"* *"ready"* *"isRun"* *"active"*	*"M (0)"* *"M (1)"* *"M (2)"*	*"free"* *"busy*	*"free"* *"busy"*	*"free"* *"busy*	*"true"* *"false"*	*"true"* *"false"*	*"true"* *"false"*

Table 5: Average test suite sizes and generation times for PLC (p = 9), with t varied up to 8.

t	the size of model	TPA Size/Time(s)	GS Size/Time(s)	PSO Size/Time(s)	TLBO Size/Time(s)	BA Size/Time(s)
2	5_process_3_memory	42.18/0.06	42.07/1.02	42.03/1.02	41.98/1.52	42.10/1.04
	6_process_3_memory	49.62/0.05	50.29/1.25	50.84/1.27	50.58/1.83	50.01/1.22
3	5_process_3_memory	130.11/0.22	130.51/5.27	132.5/5.13	129.84/7.50	131.02/5.06
	6_process_3_memory	151.51/0.28	149.05/6.01	148.96/5.72	151.53/9.15	151.07/6.39
4	5_process_3_memory	354.15/1.03	355.13/23.19	355.21/22.53	358.93/36.74	355.10/24.07
	6_process_3_memory	425.73/1.35	427.92/28.86	425.43/27.09	432.81/43.24	426.97/27.75
5	5_process_3_memory	779.71/2.83	775.41/60.8	777.93/53.91	782.51/84.69	781.94/29.03
	6_process_3_memory	934.18/3.71	936.13/73.08	938.04/65.17	941.43/89.61	937.03/34.73
6	5_process_3_memory	1641.85/4.93	1654.37/87.60	1649.05/82.47	1653.94/123.35	1645.17/83.71
	6_process_3_memory	1971.42/7.15	1985.72/104.64	1974.29/99.93	1973.34/152.69	1971.79/99.83
7	5_process_3_memory	3119.31/5.42	3116.52/84.53	3115.26/82.60	3127.52/119.83	3116.93/84.41
	6_process_3_memory	3736.18/7.17	3749.29/105.77	3738.52/101.30	3761.35/144.42	3741.30/101.07
8	5_process_3_memory	4660.82/6.78	4671.37/92.39	4678.10/86.30	4673.04/123.06	4669.30/89.83
	6_process_3_memory	4683.63/5.34	4684.54/108.74	4693.53/105.93	4691.83/154.20	4693.04/110.43

OSS models the process of shopping by customers in a store through the Internet such that customers firstly view the list of products, then select and order some of them, and finally pay the bill through a credit card. In this system, if all customers have finished their shopping, a deadlock state occurs. Table 6 shows the parameters name along with their values for a sample model of this case study with 4 customers and 8 goods. According to this table, this model has 8 parameters with different values of $d_1 = 4$, $d_2 = 2$, $d_3 = 2$, $d_4 = 4$, $d_5 = 2$, $d_6 = 8$, $d_7 = 2$, and $d_8 = 2$. So, the covering array for this model will be CA (N; t, 8, $4^2 8^1 2^5$).

Table 7 displays average test suite sizes and generation times for two different models of OSS. This case study has eight parameters (i.e. $p = 8$) and interaction strength (i.e. t) is between 2 and 7. As shown in this table, TPA outperforms the others in terms of performance and efficiency in most configurations.

Table 6: All parameters along with their values for a OSS's model with 4 customers and 8 goods

Parameter Name	Customer Name	Shop Status	Cart Name	Bill Name	Cart Status	Good Name	Good Status	Paid Status
Parameter Values	*"Cus* (0)" *"Cus* (1)" *"Cus* (2)" *"Cus* (3)"	*"true"* *"false"*	*"Cart* (0)" *"Cart* (1)"	*"Bill* (0)" *"Bill* (1)" *"Bill* (2)" *"Bill* (3)"	*"free"* *"busy*	*"Good* (0)" *"Good* (1)" *"Good* (2)" *"Good* (3)" *"Good* (4)" *"Good* (5)" *"Good* (6)" *"Good* (7)"	*"Bought"* *"noBought"*	*"true"* *"false"*

Table 7: Average test suite sizes and generation times for OSS ($p = 8$), with t varied up to 7.

t	**the size of model**	**TPA Size/Time(s)**	**GS Size/Time(s)**	**PSO Size/Time(s)**	**TLBO Size/Time(s)**	**BA Size/Time(s)**
2	4_cus_8_good	34.12/0.42	32.21/0.63	32.32/0.70	32.66/1.21	32.33/0.68
	5_cus_8_good	40.23/0.42	41.64/0.89	40.16/0.86	41.92/1.36	41.29/0.79
3	4_cus_8_good	135.33/0.56	138.03/4.01	138.66/4.31	138.21/5.51	143.66/4.03
	5_cus_8_good	168.84/069	178.31/5.15	177.00/5.30	172.31/6.92	174.55/5.09
4	4_cus_8_good	335.05/0.97	332.35/14.12	334.73/13.40	342.95/19.45	340.92/14.16
	5_cus_8_good	405.75/0.98	402.23/15.46	397.32/15.64	419.43/2.68	401.31/15.51
5	4_cus_8_good	787.25/184	788.82/30.66	791.83/28.31	794.12/40.93	790.41/28.96
	5_cus_8_good	978.46/1.85	987.72/37.43	994.54/35.48	984.32/54.59	985.41/34.50
6	4_cus_8_good	1555.23/2.31	1561.34/36.11	1556.37/33.64	1559.42/47.57	1561.47/34.54
	5_cus_8_good	1942.85/2.49	1946.32/45.53	1952.45/44.42	1958.10/66.32	1959.11/46.41
7	4_cus_8_good	2365.85/3.63	2388.63/39.84	2411.13/26.67	2426.72/34.35	2414.64/36.20
	5_cus_8_good	2987.75/2.90	2993.53/28.45	2997.12/37.41	3033.51/47.86	2991.51/36.69

FWS describes a network system in which firewalls play the role of barriers to prevent unauthorized access to secured and controlled internal networks. In fact, a firewall separates the network into internal and external parts. In the external part, safe or unsafe packets can be generated, whereas only safe packets can be produced in the internal part. These packets (safe or unsafe) can move from the external part to the internal part, and the firewall should stop the unsafe packets and denies them. In such situation, it is called FWS has faced with a deadlock state. Table 8 displays the parameters name along with their values for a sample model of this case study with 6 in-locations (LI) and 6 out-locations (LO). As seen, this model has 6 parameters with different values of $d_1 = 6$, $d_2 = 2$, $d_3 = 6$, $d_4 = 2$, $d_5 = 2$, and $d_6 = 5$. Therefore, the covering array for this model will be CA (N; t, 6, $6^2 5^1 2^3$).

In Table 9, average test suite sizes and generation times are displayed for three different models of FWS. This case study has 6 parameters (i.e. $p = 6$) and interaction strength (i.e. t) is between 2 and 5. Due to this table, TPA has

the best performance in all configurations. Exception two configurations, TPA has the higher (or equal) efficiency compared to the others.

Table 8: All parameters along with their values for a FWS's model with 6 in-locations (LI) and 6 out-locations (LO)

Parameter Name	LI Name	isFound	LO Name	hasSafeP	hasunSafeP	Position
Parameter Values	*"LI* (0)" *"LI* (1)" *"LI* (2)" *"LI* (3)" *"LI* (4)" *"LI* (5)"	*"true"* *"false"*	*"LO* (0)" *"LO* (1)" *"LO* (2)" *"LO* (3)" *"LO* (4)" *"LO* (5)"	*"true"* *"false"*	*"true"* *"false"*	*"onLO"* *"onLI"* *"onFW"* *"onIF_LO"* *"onIF_LI"*

Table 9: Average test suite sizes and generation times for FWS (p = 6), with t varied up to 5.

t	the size of model	TPA Size/Time(s)	GS Size/Time(s)	PSO Size/Time(s)	TLBO Size/Time(s)	BA Size/Time(s)
2	fire_6_LI_6_LO	37.00/2.01	37.51/0.49	37.00/00.43	36.98/0.63	39.5/0.45
	fire_8_LI_8_LO	65.11/3.03	65.51/0.81	65.98/0.79	65.53/1.07	65.71/0.83
	fire_10_LI_10_LO	260.74/3.11	262.63/3.73	262.54/3.19	263.10/4.77	262,49/3.44
3	fire_6_LI_6_LO	111.23/2.04	113.51/1.75	113.92/1.48	110.97/1.87	110.53/1.44
	fire_8_LI_8_LO	193.63/3.17	195.01/6.02	195.31/5.19	196.10/9.00	195.24/5.93
	fire_10_LI_10_LO	503.51/4.52	507.46/15.33	506.93/17.12	505.24/27.30	505.00/19.16
4	fire_6_LI_6_LO	263.31/2.12	260.31/2.97	262.57/2.83	263.91/3.88	260.91/2.91
	fire_8_LI_8_LO	454.75/3.42	454.50/13.11	454.51/14.64	461.57/18.37	457.84/14.30
	fire_10_LI_10_LO	709.31/4.72	712.51/21.76	713.50/19.51	709.73/25.94	712.00/20.11
5	fire_6_LI_6_LO	468.21/2.21	472.80/3.15	481.31/3.79	471.92/5.55	474.10/4.08
	fire_8_LI_8_LO	840.90/3.66	846.03/13.70	841.52/18.81	848.83/22.65	847.50/14.91
	fire_10_LI_10_LO	1296.41/4.23	1305.31/22.78	1314.35/24.92	1301.41/32.33	1307/25.64

HBS deals with a system by which the clients can perform the hotel booking. To do this, HBS has some agents that receive and process the requests of the clients. In this system, if all clients have performed their booking, a deadlock state occurs. Table 10 shows the parameters name along with their values for a sample model of this case study. According to this table, this model has 13 parameters with different values of $d_1 = 3$, $d_2 = 2$, $d_3 = 9$, $d_4 = 8$, $d_5 = 2$, $d_6 = 19$, $d_7 = 3$, $d_8 = 2$, $d_9 = 1$, $d_{10} = 2$, $d_{11} = 19$, $d_{12} = 1$, and $d_{13} = 1$. As a result, the covering array for this model will be CA (N; t, 13, $2^4 9^1 3^2 (19)^2 1^3 8^1$).

In Table 11, average test suite sizes and generation times are displayed for one sample model of HBS. This case study has thirteen parameters (i.e. p = 13) and interaction strength (i.e. t) is between 2 and 8. As seen in the table, similar to the previous case studies TPA outperforms the others in terms of performance in all configurations. At t = 2, all approaches have equal efficiency. In most configurations, TPA outperforms the others in terms of efficiency.

In the manually way, in addition to TPA and GS, we use TConfig as a mathematic method, PICT and IPOG as greedy algorithms, CS, PSTG, and DPSO as evolutionary algorithms. Table 12 compares TPA with the others for $6 \leq t \leq 25$. In this table, only average test suite sizes are shown. As can be seen, the methods CS, TConfig, IPOG and PSTG can only produce test suites up to t = 6. DPSO can generate test suites up to t = 12. Whereas, methods PICT and GS can generate test suites up to t = 16 and t = 20 respectively. According to the table, TPA has the best efficiency among all mentioned methods. It can support interaction strength up to t = 25.

Table 10: All parameters along with their values for a HBS's model

Parameter Name	Customer Name	Room Number	UnPaid Amount	Paid Amount	CreditC Number	Bill No	Room Status
Parameter Values	*"Andre"* *"Wegener"* *"Coburg"*	2 1	0 20000 40000 …. 160000	0 20000 40000 …. 140000	"667540" "187331"	1000 1001 … 1018	"*registerd*" "*booked*" "*vacant*"
Parameter Name	isFound	Phone	Occupied	Bill_Cntr	Amount	Hotel Location	
Parameter Values	*"true"* *"false"*	"+4995618210"	*"true"* *"false"*	1001 1002 …. 1019	20000	*"Austria"*	

Table 11: Average test suite sizes and generation times for a sample model of HBS (p = 13), with t varied up to 8.

t	**TPA Size/Time(s)**	**GS Size/Time(s)**	**PSO Size/Time(s)**	**TLBO Size/Time(s)**	**BA Size/Time(s)**
2	23.33/0.88	23.45/1.24	23.83/1.23	23.11/1.72	23.42/1.29
3	84.41/2.76	85.52/11.85	86.28/10.94	87.42/12.33	87.41/10.98
4	277.54/8.33	285.74/105.51	283.04/107.34	293.41/110.12	287.10/109/74
5	831.03/66.21	841.22/674.49	832.45/740.31	845.02/699.39	853.84/752.88
6	2165.30/204.45	2154.75/2345.91	2173.03/2639.47	2194.63/2503.31	2199.12/2893.48
7	5010.31/598.61	5001.87/7169.65	5012.34/8042.93	5023.03/11084.21	5020.41/12648.31
8	10187.24/973.54	10207.24/13764.29	10205.36/15069.32	11842.73/25437.60	10928.62/23841.10

As mentioned before, *MutationRate* specifies the mutation probability of the selected genes in the *q*-mutation operator. Due to the functionality of this operator, large values for *MutationRate* concludes the best results. To examine this assertion, we have executed TPA on CA (N; t, 2^{10}), $2 \leq t \leq 6$ for different values of *MutationRate*. The chart of Fig. 9 show generated test suites sizes for 10 independent runs. According to this chart, TPA has the best efficiency for *MutationRate* = 0.8.

Similar to *MutationRate*, *maxGenerations* is also an effective parameter on the functionality of TPA. This parameter determines the maximum iteration number of *q*-mutation operator. To check the impact of this parameter, we have executed TPA on CA (N; t, 3^7), $2 \leq t \leq 6$ for different values of *maxGenerations*. The chart of Fig. 10 illustrates generated test suites sizes for 10 independent runs. As seen in the chart, whatever the value of this parameter is high, *q*-mutation operator is more repeated and this causes the weight of the current test case is increased. Hence, small values for this parameter have the negative effect on the efficiency of TPA. When its value increments, the efficiency also raises. TPA has the best efficiency for *maxGenerations* = 100. Off course, large values (>100) don't change the efficiency.

Table 12: Average test suite sizes for higher strengths

CA	Evolutionary algorithms					Mathematic method	Greedy algorithms	
	TPA *N*	GS *N*	CS *N*	PSTG *N*	DPSO *N*	TConfig *N*	PICT *N*	IPOG *N*
CA (*N*; 6, 3^8)	1401.46	1405.9	1399	1401	1409	1515	1455	1409
CA (*N*; 7, 3^9)	4444.00	4444.9	NS	NS	4451	> day	4618	NS
CA (*N*; 8, 3^{10})	13923.91	13921.4	NS	NS	13933	> day	14599	NS
CA (*N*; 9, 3^{11})	43529.31	43835	NS	NS	> day	> day	45521	NS
CA (*N*; 10, 3^{12})	115393.10	136153.30	NS	NS	> day	> day	141990	NS
CA (*N*; 11, 3^{14})	266933.39	267196.30	NS	NS	> day	> day	278993	NS
CA (*N*; 12, 2^{14})	8874.73	8898.0	NS	NS	8972	> day	9112	NS
CA (*N*; 13, 2^{14})	9894.41	10272.0	NS	NS	> day	> day	12441	NS
CA (*N*; 14, 2^{15})	23437.44	23389.6	NS	NS	> day	> day	25036	NS
CA (*N*; 15, 2^{16})	45452.39	46698.6	NS	NS	> day	> day	51127	NS
CA (*N*; 16, 2^{17})	95653.12	95709.3	NS	NS	> day	> day	100266	NS
CA (*N*; 17, 2^{18})	177312.93	179595.6	NS	NS	> day	> day	> day	NS
CA (*N*; 18, 2^{19})	356156.30	330463.0	NS	NS	> day	> day	> day	NS
CA (*N*; 19, 2^{20})	637322.19	625001.6	NS	NS	> day	> day	> day	NS
CA (*N*; 20, 2^{20})	1048576	1048576	NS	NS	> day	> day	> day	NS
CA (*N*; 21, 2^{21})	2097152	NS	NS	NS	> day	> day	> day	NS
CA (*N*; 22, 2^{22})	4194303	NS	NS	NS	> day	> day	> day	NS
CA (*N*; 23, 2^{23})	8388607	NS	NS	NS	> day	> day	> day	NS
CA (*N*; 24, 2^{24})	16777215	NS	NS	NS	> day	> day	> day	NS
CA (*N*; 25, 2^{25})	33554431	NS	NS	NS	> day	> day	> day	NS

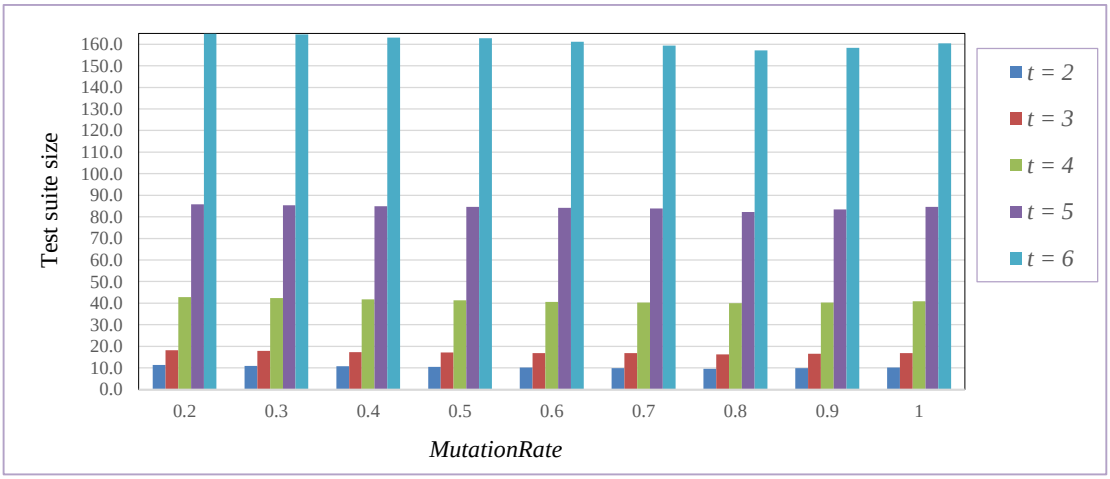

Fig. 9. The effect of *MutationRate* on the efficiency of TPA

6.1. Statistical analysis of results

In this section, we employ the Friedman test [68] for multiple comparisons of all algorithms, and the Wilcoxon signed-rank [69] for pairwise comparisons of TPA and each of the others.

The Friedman test is a non-parametric statistical hypothesis test, which can be employed for multiple comparisons of several related samples. This test ranks the algorithms for each configuration separately and then computes mean ranks over all configurations. The first rank is given to an algorithm with the least mean rank (i.e. the best efficiency), the second rank to another algorithm with the second best and so on. If two algorithms are given similar ranks, it will be concluded that they have similar efficiency. This test has two output test statistics: (1) Chi-

Square which is like a variance over the mean ranks; whatever the mean ranks are far from each other, it's value is more gerater than 0. (2) Asymp. Sig. which is a *p*-value. If *p* is less than 0.05, the null hypothesis can be rejected and this shows that there is a meaningful difference among different groups of data. In here, the Friedman test is carried out using the SPSS toolbox on the results displayed in Tables 3, 5, 7, 9, 11, and 12 (overall, 57 different configurations). According to Table 13, it is concluded that (1) TPA obtains the first rank (the best efficiency) among the considered algorithms. (2) The mean ranks are far from each other (Chi-Square = 57.664). (3) There is a significant difference among the performance of algorithms (Asymp. Sig. = 0.000).

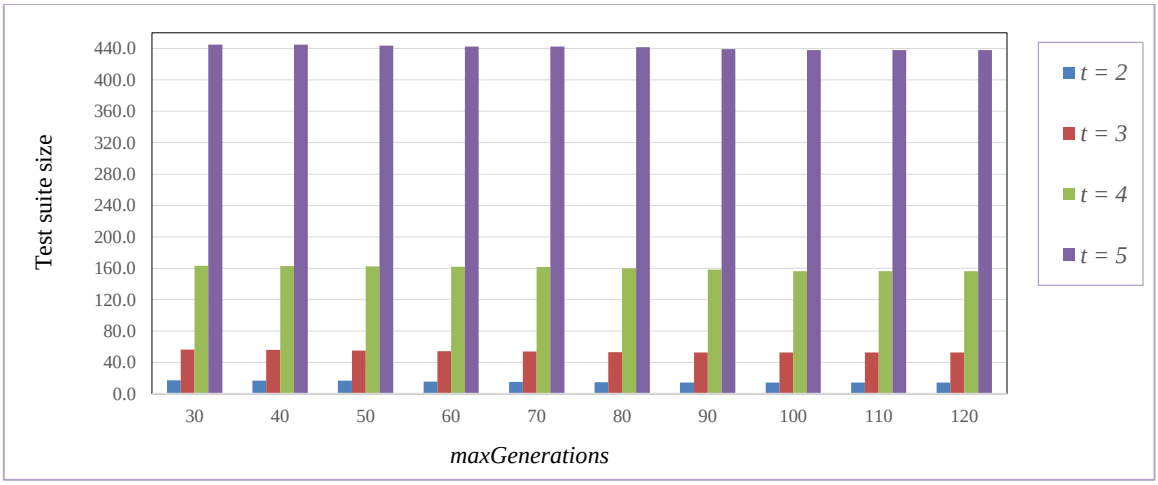

Fig. 10. The effect of *maxGenerations* on the efficiency of TPA

Table 13: Results of the Friedman test

	TPA	GS	PSO	TLBO	BA
Mean Rank	1.92	2.68	3.22	3.75	3.43
Rank	1	2	3	5	4
Chi-Square = 57.664 , Asymp. Sig. = 0.000					

Wilcoxon signed-rank is a non-parametric statistical hypothesis test which can be employed for pair comparisons of two related samples. Similar to the Friedman test, this test can also be performed using the SPSS toolbox. For two given algorithms A and B, this test has the following output test statistics: **(1)** R^- /R^+ shows the number of samples in which the algorithm A outperforms/ underperforms the algorithm B. **(2)** $R^=$ denotes the number of samples in which A and B have the equal efficiency. **(3)** Asymp. Sig. which is a *p*-value; if *p* is less than 0.05, it is concluded that A is significantly better than B. Table 14 shows the Wilcoxon signed rank test results for pair comparisons of TPA with GS, PSO, TLBO, and BA. For each pair comparison, R^-, R^+, $R^=$, and *p*-value are reported. According to this table, the values of R^- are much greater than the ones of R^+ and $R^=$ in all pair comparisons. Hence, we conclude that TPA has the best efficiency. Moreover, all obtained values of *p* are less than 0.05, and this confirms that TPA is meaningfully better than the other algorithms.

Table 14: Results of the Wilcoxon signed-rank test

	TPA-GS	TPA-PSO	TPA-TLBO	TPA-BA
$\mathbf{R^-}$	39	41	49	46
$\mathbf{R^+}$	18	15	8	11
$\mathbf{R^=}$	0	1	0	0
Asymp. Sig. (2-tailed)	$p < 0.05$	$p < 0.05$	$p < 0.05$	$p < 0.05$

7. Threats to validity

In this section, we discuss about the possible threats to validity of our work. The first threat is related to the first and second phases of TPA. It is probable that TPA cannot perform the two first phases successfully. In other words, TPA may not detect a deadlock state in the given model, and so the parameters and their values will not ready to start the third phase, i.e., genearting the minimum test suite. The second threat is that the impartiality of benchmark experiments may be a concern. As mentioned in Section 6, TPA and other approaches are evaluated in the automatically and manually ways that parameters and their values are prepared. In the automatically way, TPA is compared with GS, TLBO, BA, and PSO, which all of them have been implemented in GROOVE. Whereas, in the manually way, TPA is compared with TConfig, PICT, IPOG, CS, PSTG, and DPSO, which their source codes are not available. Therefore, in the manually way, unlike the automatically way, there may be a concern because the comparisons rely only on the previous published results. The third threat is related to how each approach implements the coverage matrix and calculates the weight of a test case. In the automatically way, these isn't any concern about these issues because of implementing all approaches in GROOVE. But, in the manually way, there is a concern about these subjects due to a lack of source codes of the TConfig, PICT, IPOG, CS, PSTG, and DPSO approaches.

8. Conclusion and Future Works

In this paper, we discussed about the existing challenges of *t*-way strategy and proposed solutions to resolve them. The first and second challenges respectively relate to the low quality of the generated TS (i.e. some complex errors may not be found through this TS) and how to prepare parameters and their values for the *t*-way strategy. Moreover, the third challenge is the low generation speed and the large size of the generated test suite. To resolve these challenges, we proposed a three-phase approach (so-called TPA). In the first phase, TPA used an optimized version of model checking (OMC) to extract the information about special errors from a model of SUT and injecting them into the TS. In the second phase, TPA uses the explored states in the first phase to prepare parameters and their values automatically. To handle the third challenge, several methods using evolutionary algorithms have been proposed. Although some of them can support test suite generation up to $t = 20$, they have the low generation speed. In the third phase, TPA applies an adopted version of evolution strategy to increase the generation speed. Moreover, TPA can generate test suites with the interaction strength up to $t = 25$. Experimental results confirm that TPA outperforms the other greedy, mathematic, and evolutionary algorithms in terms of test suite size and generation

speed. Studing other evolutionary algorithms and proposing a hybrid approach of them and TPA can be considered as a future work.

Compliance with Ethical Standards

Conflict of interest The authors declare that there is no conflict of interest for any of them in this article.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent Informed consent was obtained from all individual participants included in the study.

Authors contributions SE was responsible for conceptualization, methodology, software and writing original draft; VR and EP were responsible for supervision, writing, reviewing and editing.

References

[1] B. S. Ahmed, T. S. Abdulsamad, and M. Y. Potrus, "Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the cuckoo search algorithm," *Inf. Softw. Technol.*, vol. 66, pp. 13–29, 2015.

[2] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *Int. J. Embed. Syst. Appl. IJESA*, vol. 2, no. 2, pp. 29–50, 2012.

[3] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*, vol. 2. Wiley Online Library, 2004.

[4] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[5] M. Grindal and J. Offutt, "Input parameter modeling for combination strategies," in *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, 2007, pp. 255–260.

[6] S. Esfandyari and V. Rafe, "Extracting Combinatorial Test parameters and their values using model checking and evolutionary algorithms," *Appl. Soft Comput.*, vol. 91, p. 106219, 2020.

[7] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, 2004, pp. 72–77.

[8] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, "Constructing a t-way interaction test suite using the particle swarm optimization approach," *Int. J. Innov. Comput. Inf. Control*, vol. 8, no. 1, pp. 431–452, 2012.

[9] H. Kastenberg and A. Rensink, "Model checking dynamic states in GROOVE," in *International SPIN Workshop on Model Checking of Software*, 2006, pp. 299–305.

[10] R. Sagarna and J. A. Lozano, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms," *Eur. J. Oper. Res.*, vol. 169, no. 2, pp. 392–412, 2006.

[11] R. Santana, P. Larrañaga, and J. A. Lozano, "Protein folding in simplified models with estimation of distribution algorithms," *IEEE Trans. Evol. Comput.*, vol. 12, no. 4, pp. 418–438, 2008.

[12] E. Pira, "A novel approach to solve AI planning problems in graph transformations," *Eng. Appl. Artif. Intell.*, vol. 92, p. 103684, 2020.

[13] B. Yuan, M. Orlowska, and S. Sadiq, "Finding the optimal path in 3D spaces using EDAs–the wireless sensor networks scenario," in *International Conference on Adaptive and Natural Computing Algorithms*, 2007, pp. 536–545.

[14] M. Pelikan, D. E. Goldberg, and S. Tsutsui, "Hierarchical Bayesian optimization algorithm: toward a new generation of evolutionary algorithms," in *SICE 2003 Annual Conference (IEEE Cat. No. 03TH8734)*, 2003, vol. 3, pp. 2738–2743.

[15] G. Rozenberg, *Handbook of Graph Grammars and Comp.*, vol. 1. World scientific, 1997.

[16] G. Taentzer, "AGG: A graph transformation environment for modeling and validation of software," in *International Workshop on Applications of Graph Transformations with Industrial Relevance*, 2003, pp. 446–453.

[17] E. Pira, “Using Markov Chain Based Estimation of Distribution Algorithm for Model-Based Safety Analysis of Graph Transformation,” *J. Comput. Sci. Technol.*, vol. 36, pp. 839–855, 2021.
[18] E. Pira, V. Rafe, and A. Nikanjam, “Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations,” *Inf. Softw. Technol.*, vol. 97, pp. 110–134, 2018.
[19] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.
[20] E. Pira, M. Z. Reza, and F. Soltani, “Verification of confliction and unreachability in rule-based expert systems with model checking,” *Int. J. Artif. Intell. Appl.*, vol. 5, no. 2, p. 21, 2014.
[21] S. Esfandyari and V. Rafe, “A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy,” *Inf. Softw. Technol.*, vol. 94, pp. 165–185, 2018.
[22] C. Nie, H. Wu, X. Niu, F.-C. Kuo, H. Leung, and C. J. Colbourn, “Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures,” *Inf. Softw. Technol.*, vol. 62, pp. 198–213, 2015.
[23] A. Hartman and L. Raskin, “Problems and algorithms for covering arrays,” *Discrete Math.*, vol. 284, no. 1–3, pp. 149–156, 2004.
[24] N. Kobayashi, “Design and evaluation of automatic test generation strategies for functional testing of software,” *Osaka Jpn. Osaka Univ*, 2002.
[25] A. Hartman, “Software and hardware testing using combinatorial covering suites,” in *Graph theory, combinatorics and algorithms*, Springer, 2005, pp. 237–266.
[26] A. W. Williams and R. L. Probert, “A practical strategy for testing pair-wise coverage of network interfaces,” in *Proceedings of ISSRE’96: 7th International Symposium on Software Reliability Engineering*, 1996, pp. 246–254.
[27] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Comput. Surv. CSUR*, vol. 43, no. 2, pp. 1–29, 2011.
[28] Y. Lei and K.-C. Tai, “In-parameter-order: A test generation strategy for pairwise testing,” in *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*, 1998, pp. 254–261.
[29] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, “IPOG: A general strategy for t-way software testing,” in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’07)*, 2007, pp. 549–556.
[30] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, “Refining the in-parameter-order strategy for constructing covering arrays,” *J. Res. Natl. Inst. Stand. Technol.*, vol. 113, no. 5, p. 287, 2008.
[31] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The AETG system: An approach to testing based on combinatorial design,” *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, 1997.
[32] J. Czerwonka, “Pairwise testing in the real world: Practical extensions to test-case scenarios,” *Microsoft Corp. Softw. Test. Tech. Artic.*, 2008.
[33] K. Z. Zamli, M. F. Klaib, M. I. Younis, N. A. M. Isa, and R. Abdullah, “Design and implementation of a t-way test data generation strategy with automated execution tool support,” *Inf. Sci.*, vol. 181, no. 9, pp. 1741–1758, 2011.
[34] J. Stardom, *Metaheuristics and the search for covering and packing arrays*. Simon Fraser University Burnaby, 2001.
[35] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, “A variable strength interaction testing of components,” in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, 2003, pp. 413–418.
[36] T. Mahmoud and B. S. Ahmed, “An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use,” *Expert Syst. Appl.*, vol. 42, no. 22, pp. 8753–8765, 2015.
[37] K. Z. Zamli, B. S. Ahmed, T. Mahmoud, and W. Afzal, “Fuzzy Adaptive Tuning of a Particle Swarm Optimization Algorithm for Variable-Strength Combinatorial Test Suite Generation,” *ArXiv Prepr. ArXiv181005824*, 2018.
[38] B. S. Ahmed, K. Z. Zamli, and C. P. Lim, “Application of particle swarm optimization to uniform and variable strength covering array construction,” *Appl. Soft Comput.*, vol. 12, no. 4, pp. 1330–1347, 2012.
[39] Y. A. Alsariera, A. H. Al Omari, M. A. Albawaleez, Y. K. Sanjalawe, and K. Z. Zamli, “Hybridized BA & PSO t-way Algorithm for Test Case Generation,” *IJCSNS*, vol. 21, no. 10, p. 343, 2021.
[40] H. Wu, C. Nie, F.-C. Kuo, H. Leung, and C. J. Colbourn, “A discrete particle swarm optimization for covering array generation,” *IEEE Trans. Evol. Comput.*, vol. 19, no. 4, pp. 575–591, 2014.

[41] A. R. A. Alsewari and K. Z. Zamli, "Design and implementation of a harmony-search-based variable-strength t-way testing strategy with constraints support," *Inf. Softw. Technol.*, vol. 54, no. 6, pp. 553–568, 2012.
[42] K. Z. Zamli, F. Din, S. Baharom, and B. S. Ahmed, "Fuzzy adaptive teaching learning-based optimization strategy for the problem of generating mixed strength t-way test suites," *Eng. Appl. Artif. Intell.*, vol. 59, pp. 35–50, 2017.
[43] P. Flores and Y. Cheon, "PWiseGen: Generating test cases for pairwise testing using genetic algorithms," in *2011 IEEE International Conference on Computer Science and Automation Engineering*, 2011, vol. 2, pp. 747–752.
[44] J. D. McCaffrey, "An empirical study of pairwise test set generation using a genetic algorithm," in *2010 Seventh International Conference on Information Technology: New Generations*, 2010, pp. 992–997.
[45] P. Bansal, S. Sabharwal, S. Malik, V. Arora, and V. Kumar, "An approach to test set generation for pair-wise testing using genetic algorithms," in *International Symposium on Search Based Software Engineering*, 2013, pp. 294–299.
[46] K. M. Htay, R. R. Othman, A. Amir, and J. M. H. Alkanaani, "Gravitational search algorithm based strategy for combinatorial t-way test suite generation," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 8, pp. 4860–4873, 2022.
[47] H. N. N. Al-Sammarraie and D. N. Jawawi, "Multiple black hole inspired meta-heuristic searching optimization for combinatorial testing," *Ieee Access*, vol. 8, pp. 33406–33418, 2020.
[48] J. M. Altmemi, R. R. Othman, and R. Ahmad, "SCAVS: Implement Sine Cosine Algorithm for generating Variable t-way test suite," in *IOP Conference Series: Materials Science and Engineering*, 2020, vol. 917, no. 1, p. 012011.
[49] D. Karaboga, "Artificial bee colony algorithm," *scholarpedia*, vol. 5, no. 3, p. 6915, 2010.
[50] A. K. Alazzawi, H. M. Rais, and S. Basri, "Artificial bee colony algorithm for t-way test suite generation," in *2018 4th International Conference on Computer and Information Sciences (ICCOINS)*, 2018, pp. 1–6.
[51] A. K. Alazzawi, H. M. Rais, and S. Basri, "ABCVS: An artificial bee colony for generating variable t-way test sets," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 4, 2019.
[52] A. K. Alazzawi, H. M. Rais, and S. Basri, "Hybrid artificial bee colony algorithm for t-way interaction test suite generation," in *Software Engineering Methods in Intelligent Algorithms: Proceedings of 8th Computer Science On-line Conference 2019, Vol. 1 8*, 2019, pp. 192–199.
[53] S. Esfandyari and V. Rafe, "GALP: a hybrid artificial intelligence algorithm for generating covering array," *Soft Comput.*, vol. 25, no. 11, pp. 7673–7689, 2021.
[54] H. L. Zakaria, K. Z. Zamli, and F. Din, "Hybrid migrating birds optimization strategy for t-way test suite generation," in *Journal of Physics: Conference Series*, 2021, vol. 1830, no. 1, p. 012013.
[55] J. B. Odili, A. B. Nasser, A. Noraziah, M. H. A. Wahab, and M. Ahmed, "African buffalo optimization algorithm based t-way test suite generation strategy for electronic-payment transactions," in *Proceedings of International Conference on Emerging Technologies and Intelligent Systems: ICETIS 2021 (Volume 1)*, 2022, pp. 160–174.
[56] C. Luo *et al.*, "AutoCCAG: An automated approach to constrained covering array generation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 201–212.
[57] E. Pira, V. Rafe, and S. Esfandyari, "Minimum Covering Array Generation Using Success-History and Linear Population Size Reduction based Adaptive Differential Evolution Algorithm".
[58] M. Ahmed, A. B. Nasser, and K. Z. Zamli, "Construction of Prioritized T-Way Test Suite Using Bi-Objective Dragonfly Algorithm," *IEEE Access*, vol. 10, pp. 71683–71698, 2022.
[59] E. Pira, V. Rafe, and A. Nikanjam, "Using evolutionary algorithms for reachability analysis of complex software systems specified through graph transformation," *Reliab. Eng. Syst. Saf.*, vol. 191, p. 106577, 2019.
[60] E. Pira, V. Rafe, and A. Nikanjam, "Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm," *J. Syst. Softw.*, vol. 131, pp. 181–200, 2017.
[61] "Generating dynamically nested loops." https://www.codeproject.com/Tips/759707/Generating-dynamically-nested-loops (accessed Oct. 13, 2020).
[62] Z. Abbasi, S. Esfandyari, and V. Rafe, "Covering Array Generation using Teaching Learning base Optimization Algorithm," *TABRIZ J. Electr. Eng.*, vol. 48, no. 1, pp. 161–171, 2018.
[63] R. Yousefian, S. Aboutorabi, and V. Rafe, "A greedy algorithm versus metaheuristic solutions to deadlock detection in Graph Transformation Systems," *J. Intell. Fuzzy Syst.*, vol. 31, no. 1, pp. 137–149, 2016.
[64] Á. Schmidt, "Model checking of visual modeling languages," *Bp. Univ. Technol. Hung.*, 2004.

[65] E. Pira, V. Rafe, and A. Nikanjam, “Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations,” *Inf. Softw. Technol.*, vol. 97, pp. 110–134, 2018.
[66] S. M. Bellovin and W. R. Cheswick, “Network firewalls,” *IEEE Commun. Mag.*, vol. 32, no. 9, pp. 50–57, 1994.
[67] A. Kalaee and V. Rafe, “Model-based test suite generation for graph transformation system using model simulation and search-based techniques,” *Inf. Softw. Technol.*, vol. 108, pp. 1–29, 2019.
[68] M. Friedman, “A comparison of alternative tests of significance for the problem of m rankings,” *Ann. Math. Stat.*, vol. 11, no. 1, pp. 86–92, 1940.
[69] F. Wilcoxon, “Individual comparisons by ranking methods,” in *Breakthroughs in statistics*, Springer, 1992, pp. 196–202.