Applied Soft Computing Journal 91 (2020) 106219

Contents lists available at ScienceDirect

Applied Soft Computing Journal

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

Extracting Combinatorial Test parameters and their values using model checking and evolutionary algorithms

Sajad Esfandyari, Vahid Rafe *

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

ARTICLE INFO

Article history:
Received 6 July 2019
Received in revised form 19 January 2020
Accepted 6 March 2020
Available online xxxx

Keywords:
Covering array
Model checking
Genetic algorithm
GROOVE

ABSTRACT

Combinatorial Testing (CT) is one of the popular testing approaches for generating a minimum test suite to detect defects caused by interactions between subsystems. One of the most practical CT methods is the Covering Array (CA). While generating CA, there are at least two main groups of challenges. The first one is extracting information about parameters, identifying constraints and detecting interactions between subsystems automatically. In most of the existing approaches, this information is fed to the system manually which makes it difficult or even impossible for testing modern software systems. The second one is the speed and the array size. Even though most of the existing approaches are concentrated on this challenge, their results show that there is still room for improvement. In this paper, we propose an idea to cope with both challenges. At first, we represent a method to extract information about the system under test (SUT) from its model using model checking (MC) techniques. MC is a method that scans all possible states of the system for detecting errors. After that, we propose another new approach using genetic algorithm to generate the optimal CA in terms of speed and size. To evaluate the results, we implemented the proposed strategy along with several other metaheuristic algorithms in the GROOVE tool, an open toolset for designing and model checking graph transformation specifications. The results represent that the proposed strategy performs better than others.

1. Introduction

Often, large systems have a lot of parameters. These parameters may have several features. To completely test a SUT, it is necessary to check all available configurations, which is unrealistic and impossible. For example, consider a system with 9 parameters, each of which has 5 features. The system has 5^9 (1,953,125) testable configurations, all of which are very difficult to check, so there is a need for a method to select a sample among the entire configurations to identify errors, it is vital and indisputable [1].

CT is one of the test methods that uses the CA to generate a minimum test suite (minimum number of test cases) for detecting defects caused by the combination and interaction between subsystems [2]. CA by sampling a small number of test cases covers all possible combinations of a certain number of parameters. The Variable Strength CA (VSCA) and the Constraints CA (CCA) are also in the CT family, but the CA is more general.

Speed and array size are two important challenges while generating CA. The CA uses the t-way strategy to generate a test suite with a minimum number of test cases. Each row of the CA is a test case. The smaller size the array has, the more powerful the strategy is, but the time is also very important. There are many types of research to solve this problem, which are generally divided into two major categories based on artificial intelligence (AI) and pure computational. The strategy based on the Particle swarm optimization (PSO) algorithm [2], called Discrete Particle Swarm Optimization (DPSO), has the best performance among the strategies in terms of array size. Genetic strategy (GS) [3] based on GA has the highest rate among AI-based strategies and has good results in terms of array size. Also, Harmony Search Strategy (HSS) [4], Cuckoo Search (CS) [5] and Particle Swarm-based t-way Test Generator (PSTG) [6] strategies have also acceptable results in terms of time and array size among AI-based strategies. Pure computational strategies have also provided good tools such as Automatic Efficient Test Generator (AETG) [7], Test Configuration (TConfig) [8], Jenny [9], Intelligent Test Case Handler (ITCH) [10], Pairwise Independent Combinatorial Testing (PICT) [11], and In-Parameter-Order-General (IPOG) [12], which have good results in terms of time and sometimes array size. In the meantime, the TCAS tool is the best strategy in terms of time and array size. In addition, the results of the strategies in [13] most times have the best performance in array size.

* Corresponding author.
E-mail addresses: Sajad.a1367@gmail.com (S. Esfandyari), v-rafe@araku.ac.ir (V. Rafe).

https://doi.org/10.1016/j.asoc.2020.106219
1568-4946/

Another important issue is automatic extraction of parameters and their values. In [14] it is noted that finding parameters and their values is a creative process, and it cannot be completely automatic. Efforts have been made to automatically extract the values of parameters, determine the constraint and interaction between the parameters [15–17]. These solutions are usually designed from multi-stage algorithms and can be used in special cases. Also, they need to model input parameters. Hence, the second goal in this study is extracting the parameters and their values automatically. To do so, we use model checking (MC) techniques.

MC is a process in which all reachable states of a given system are generated in the form of a directed graph. The model checker searches the state space completely and examines the correctness a property. Using The MC is commonplace in the CT, and various work has been done with the combination of these two [18–20]. In these strategies, the input parameters are usually given separately to the CT, and after the test suite is generated, the test is performed on the SUT.

Setting the input values manually is an error prone task. Therefore, the input parameters are modeled to reduce error and increase quality, and it is called the Input Parameter Model (IPM). Using IPM is very common in CT [15,17,21]. IPM increases test quality in CT [15,22], but it is difficult to detect the values for dynamic parameters. Also, if there is a model available from the SUT, for the general test, we need two models: one for the SUT, and another one for the IPM, which both cases are error prone.

Extracting the values of the parameters from the model is very important in the domain of software testing, and was strongly addressed in [23,24]. In these solutions, the information of the parameters was extracted using a multi-stage algorithm. These approaches can only support simple cases. For example, for an array, only three values of "empty", "one-element" and "several-element" were produced. Or in [16] the requirements are firstly modeled in the UML sequence diagram, then all parameters and their values are extracted from this diagram. For example, CA (N; t, 5, 3) represents a system that has 5 variables, each of which accepts 3 attributes. But we plan to get a model of SUT as an input rather than extracting numbers manually, which is an error prone task. Then, using the MC tools, we generate the state space and, by scanning the state space, the values are automatically extracted with 100% accurately and placed in the CA.

In this paper, we first design a model based on SUT in the GROOVE model checker tool [25] and then the state space is generated. By traversing through the state space, we extract all the information needed by the CA. Then arrays are generated using metaheuristic algorithms such as PSO, BAT, TLBO, and GA, and then we will use a simple but practical method to minimize the number of test cases in a test suite. The evaluation results indicate that GA has far better results than other algorithms. With some modifications on the GA in [3] we can generate the test suite with the suitable power and speed. To prove our claim, we compared the proposed algorithm with superior strategies in the CA field. The results show that the proposed algorithm produces appropriate results.

The rest of this paper is organized as follows. Section 2 surveys state of the art. Section 3 describes the MC, the GROOVE tool and CA. In Section 4, the proposed approach is described in details. Section 5 evaluates the results. Finally, Section 6 concludes the paper and proposes some future works.

2. State of the art

There are many types of research for solving CA that are divided into two main groups [6]: (1) Mathematical methods and (2) Computational methods.

Mathematical methods are derived from some of the structures of the mathematical functions of Orthogonal Arrays (OA), but computational methods mainly use greedy strategies or over-exploitation methods to generate CAs in search space. Mathematical methods generate CA only for specific configurations [26]. One can mention the strategies available in mathematical methods to the Combinatorial Test Services [27] and TConfig [8], which are based on OA. The major problem with OA is that it has the ability to construct CAs for small and specific configurations, which makes this use more computational. The solution to most computational methods is that initially all possible combinations are generated according to input specifications. Computational methods are also divided into two general categories of pure computational and artificial intelligence-based computation [4]. Afterward, a test suite is made to cover these compounds. The main difference between different strategies is the computational method of making a test case. There are two main methods for constructing test cases in computational methods [4]: (1) One-test-at-a-time (OTAT) and (2) One-parameter-at-a-time (OPAT)

In the OTAT test, each stage produces a test case or a complete set of test samples. Then, the test case that covers most of the interactions is selected. This test case consists of one line of the final test suite (CA). The most well-known strategies are based on the AETG [7], PICT [11], TVG [28], Jenny [9] and GTWay [29].

The OPAT method introduces the CA by adding each step-by-step parameter, and then, by adding the possible combinations for the existing parameters, the steps continue until the covering completes. IPOG [12] and IPOG-D [30] are common strategies that have been developed based on this method. The first strategy that generated a test suite with OTAT is the AETG [7]. This strategy selects a number of test samples in each cycle and adds one sample test to the test suite in a greedy manner. In fact, the strategy adds a test case to the test suite that covers the largest number of interactions.

The PICT [11] strategy is also a greedy strategy for other OTAT strategies. This strategy generates all interactions and randomly selects the required test case. Due to the random basis, this strategy often produces unsatisfactory results. The strengths of this strategy are that it has the ability to generate the test suite for $t > 6$. The TVG [28] uses three T-Reduce, plus-one and Random Sets algorithms to generate the test suite. Details of these algorithms are not available, but in general, the T-Reduce algorithm has better results than two other algorithms. This strategy also generates a test suite based on OTAT.

Another strategy of OTAT is Jenny [9]. This strategy has good speed and produces relatively acceptable results for many configurations in terms of efficiency. The structure of this algorithm is such that it initially generates all 1-way interactions, then the test suite for the 2-way interactions will be completed and this procedure will continue until all t-way interactions are covered.

One of the most powerful computing strategies is the GTWay [29] strategy. This strategy, which produces a test suite based on OTAT, first stores all sample instances of the test as bit structures and uses an index table to get faster access to the components of the test cases. The strategy is one of the few strategies that produce both excellent results in both terms of performance and efficiency. GTWay strategy has the potential to produce up to $t = 12$.

Another category of methods is based on the AI algorithms. These algorithms are based on the OTAT method. With the ever-increasing use of Search Based Software Engineering (SBSE), in the optimal solution of software issues, the use of metaheuristic algorithms in CA production is more popular than other methods. Metaheuristic algorithms start with a random collection of solutions, then, in order to improve these initial solutions, a series of

changes are made to them and, finally, the best choice solution. It can be continued until all the ingredients are coated. Some of the metaheuristic algorithms that have been used so far have been SA [31,32], TS [31], GA [3,31,32], PSO [2,6,33,34], TLBO [35,36], CS [5,37], Flower Pollination [38,39], HSS [4], and Bat algorithm (BA) [40,41], that the most important of which are discussed below.

Initially, Stardom [31] implemented three SA, GA, and TS algorithms to support 2-way (up to t = 2) configurations, which in this regard, SA produced better results. Further, Cohen [32] upgraded SA to 3-WAY, while the results in this paper show that SA has better results than other algorithms. The SA strategy uses complex structures and heavy algorithms to reduce the size of the array, hence its production time is too much and therefore did not have the ability to produce high interaction strength.

The PSTG [6] strategy was the first artificial intelligence-based solution that could support 6-way. This strategy had better results than computing-based strategies, but it was weaker than SA for $t \leq 3$. The strategy also supported the VSCA. This solution has a data structure with $C(p, t) * v^t$ rows and p columns. To calculate the weight in this solution, all rows must be searched, and for faster access to the rows, the data structure is divided into $C(p, t)$ groups, and the weight of the test case is calculated using the corresponding group index. Initially, the running time for weight calculation of a test case in this method is $O(C(p, t) * v^t)$ and as more cases are covered the time tends to $O(C(p, t))$.

CS [5] was also a strong strategy that was presented after the PSTG, by changing the data structure in the PSTG, this strategy was able to generate CA in less time, but it did not have the acceptable size array of the PSTG. It also did not support VSCA and supports up to t = 6. The CS is another powerful strategy in CA production. It also uses a data structure similar to PSTG. This strategy, which uses the Cuckoo algorithm to produce the coverage array, yields far better results than PSTG and is much stronger than PSTG in terms of production time. In this solution, after covering each test case, that test case is removed from the data structure, and in the subsequent steps of weight calculation, the data structure gets smaller, which increases the speed of the strategy. One of its weak points is its inability to produce VSCA. It is also capable of producing the test suite up to the strength of 6.

The HSS [4] is based on the Harmony Search Algorithm. In this strategy, like PSTG and CS, in each iteration, a test case adds to the final test suite and the algorithm continues until the final test suite completes. This strategy can support high strength (up to t = 15). And in terms of the array size, the results are similar to those of the PSTG and CS, but its time is not available.

Another effective solution that produces the test suite for the coverage array is the Fuzzy Self Adaptive PSO (FSAPSO) [33]. This strategy is based on the PSO algorithm. In this approach, the values of parameters C1, C2 and W are fuzzy. The results published in this paper show that this strategy produces test suite up to $t = 4$ and performs better than PSTG and CS in many structures, in terms of efficiency. Due to the use of Fuzzy tools in MATLAB, this strategy is not fast enough. This solution was not capable of generating VSCA; later, the test suite with high efficiency was produced for VSCA in [34] using the PSO algorithm and the Mamdani fuzzy inference system.

The next review concerns the HHH [42] strategy, which is based on High Level Hyper-Heuristic. Instead of using a meta-heuristic algorithm, the four meta-heuristic algorithms are used in this strategy: Teaching Learning Based Optimization (TLBO), Global Neighborhood Algorithm (GNA), Particle Swarm Optimization (PSO), and Cuckoo Search Algorithm (CS). This strategy uses the Tabu algorithm; in each step, one of the four meta-heuristic algorithms is selected using the three operators improvement, diversification and intensification to generate a test case. The HHH strategy has very good results in terms of array size.

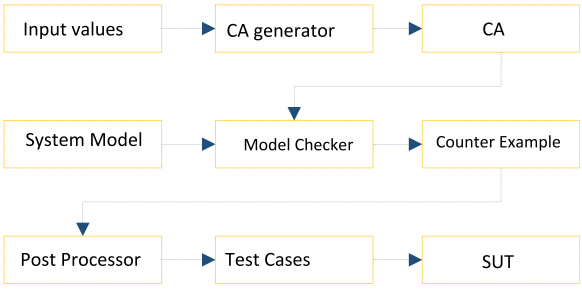

Fig. 1. Generating test suite using the CA and MC [22].

One of the weak points of the PSO-based strategy in producing the test suite is that by applying velocity, the new amount will not necessarily be improved. To overcome this problem, the DPSO [2] strategy has been able to deliver the most satisfactory results from the efficiency point of view. The strategy in the article is up to t = 4, but it can also support $t < 10$. It can be said that this strategy has the best performance in terms of the array size, but it does not have the right time to produce it. After DPSO, a lot of strategies were presented, but no significant superiority in terms of array size was due to the DPSO.

Another powerful algorithm in the field of CA is ATLBO [35]. This solution using TLBO based on Mamdani fuzzy inference system yields very good results in terms of efficiency. It can also cover VSCA. For faster access to test cases, Hash Map is used. In this study, in addition to ATLBO, the original version of TLBO is implemented and the evaluation results show that ATLBO is better than the original TLBO in terms of efficiency and performance. The results of this approach are also compared with the results of strategies DPSO, PSTG, CS and APSO, indicating that this strategy is one of the most powerful approaches in CA.

The GS [3] is also another strategy that supports CA. This strategy utilizes bit structure and changes in GA to produce appropriate results from the point of view of time relative to AI-based strategies and is able to $t = 20$ support. Unlike the CS and PSTG strategies, this solution uses a single bit for each test case, which greatly reduces the volume of data structure. Also, for faster access to the test case in weight calculation, the combinations are stored in each row, which helps increase the speed of weight calculation. The data structure in this solution is composed of $C(p, t)$ rows and v^t columns. The running time of the weight calculation for a test case is of the order $O\left(C(p, t) * v^t\right)$ which tends to $O(t)$, by covering the test cases.

The use of MC in CT is conventional. Model-checker based test generation uses a mathematical model of SUT and a MC to produce the expected results for each input. A MC can display all modes of a SUT. If a feature is not correct, the MC recognizes it and displays a counterexample. The production of the test suite by combining CA and MC is such that the input space and a model of the SUT are presented. It is created from an input using a CA constructor, and then the SUT and CA models are given to the MC. The exact steps of combining these two are shown in Fig. 1.

In general, the accuracy of entering parameters and their values in CT quality is very important. Manually entering parameter information increases the error probability. To reduce the error, the input is modeled that Called IPM. Modeling the input space is a very common practice in software testing. Generally, IPM generation methods are divided into two categories: Category Partition and Classification Trees [23]. A lot of research has been done in this area, but we will only look at a number of studies in CA. In [14], which uses the Classification Trees method, the automated generation of parameters is considered as a creative process and states that it is not possible to produce fully automatic parameters.

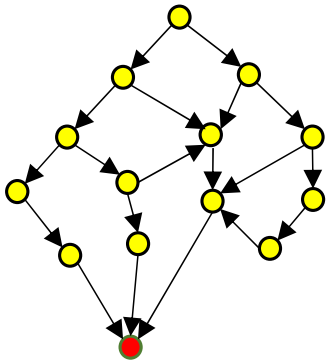

Fig. 2. Deadlock state in a state space.

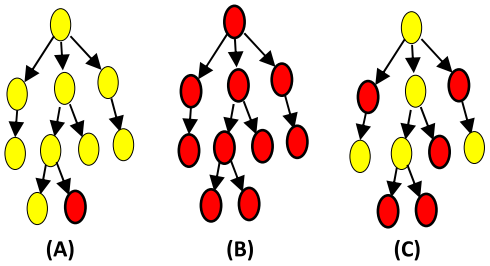

Fig. 3. Display the operators in CTL.

In [17], a strategy is presented that consists of 4 steps. In step 1 (INPUT: UML Activity Diagrams): the requirements are also modeled with the UML Activity Diagram and in step 2 (Generating XMI files) the model is converted into an XML file, and then in step 3 (CTDM Parser parses the XMI files) with three separate algorithms, the parameters, their values, and the constraints are separately detected, and ultimately in step 4 (OUTPUT: CTDM model) produces parameters and their values.

Another known strategy in CT was provided by Borazjany et al. [15]. This solution consists of two steps. One IPM and another is Input structure modeling (ISM). The ISM recognizes the structural relationships between the SUT components, and the IPM is also responsible for identifying the parameters, their values, and the constraints of each component. It also has the ability to perform unit and integration testing on the structure of the input space structure.

The next approach, the latest study of which is discussed in this paper, uses the UML Sequence Diagram to extract the parameters and their values [16]. In this solution, which is Rule-Base, requirements are firstly modeled by a UML 2.0 sequence diagram (step 1: input), then the model is converted into an XMI (XML Metadata Interchange) code (step 2: XMI code), In next (step 3: Analyzer), this solution applies the rules to the XMI code using the CT modeler tool and UML sequence diagram analyzer and produces CTDM elements, and CTDM is generated (step 4: Output).

3. Preliminaries

3.1. Model checking

The greatest amount of time and cost in designing a system is spent on system correctness. MC is a process in which all possible states of a system are created as a graph. Then, using the MC tools, all the states are checked and the system correctness is rejected or satisfied. In fact, MC is a process for evaluating or accepting a system property. When a property is not accepted for a particular batch of states, the MC provides an example of a violation (or witness) by searching the relevant state space. By examining all possible scenarios, MC determines whether the system supports the corresponding property or results in an error (for example, deadlock) [43,44]. The deadlock is referred to as state space, which has no output edges. In other words, there is a system in which there are no other scenarios for the survival of the system. In Fig. 2, the red state is deadlock.

To check properties of a system, these properties must be described in one of two types of linear temporal logic (LTL) or computational tree logic (CTL). In LTL, the future of the system is considered to be a state transition. In this logic, for each state, an output pass is considered. Formulating a system in a LTL requires a number of atomic states that are combined by two time operators X and U, and logical operators' ⌐, ∧, and v. In CTL, the future of the system is considered as a computational tree. In other words, in this logic, there may be more than one output pass for each mode. A CTL formula is comprised of sets of atomic states, A (for all possible paths), E (for at least one path), and time operators F and G (in CTL and LTL). There are LTL operators for CTL, with the difference that CTLs use two A and E operators before these operators and respectively represent the entire paths and at least one path of the graph. These operators are usually used to check out three important properties in software systems [45,46]:

- Reachability: This feature asserts that finally a particular state such as q is available. Like a deadlock. To display this property in CTL, use E <> q (Fig. 3(A). q is the red state).
- Safety: This feature claims that a bad state never happens. The CTL is used to represent the safety of A [] q (Fig. 3(B)).
- Liveness: This feature claims that a good state should finally happen. Liveness is divided into two general modes conditional and unconditional. For example, in the Dining Philosopher Problem (DPP) in conditional mode, we will say: "If a philosopher is hungry, he must eventually eat" and in an unconditional mode, we say, "Finally, a philosopher must eventually eat". The path formula A <> q assuming that q is a state property in CTL, it can be used to express survival (Fig. 3(C)).

3.2. Graph transformation system

In the MC process, at first, the system must be described by a formal modeling language. The textual and graphical modeling language is two types of formal modeling languages [47]. In text modeling languages, such as Alloy [48], standard reserved words, along with natural language parameters or phrases, are used to describe the meaning of the system. However, graphs are used to present concepts and their relationship in a graphical modeling language. This language uses graphs and graph transformations for the formal description of states and system behaviors [45].

Def. 1 (*Graph Transformation System*)**.** The graph transformation system is defined as a triple (TG, HG, R), which we will continue to examine each one [45].

- TG is a Type Graph. This graph is a tuple of TG = (TGN, TGE, src, trg), in which, TGN (Type Graph Node) is a set of node types, TGE (Type Graph) is a set of edges and src and trg of two functions as src, trg: TGE → TGN that specify the source and destination nodes.
- HG is Host Graph. It determines the initial state of the system. It should be a sample of TG, in the sense that there

must be a homogeneous graph from HG to TG so that each edge and node in the host graph is mapped to an edge type and a node type in the TG.
- R is a set of transformation rules. A transformation rule for p: LHS → RHS is composed of a pair of LHS and RHS sample graphs on the TG. The left side of the LHS represents the pre-conditions of the rule and the right side of the RHS represents the post-condition of the rule. The left and right sides of the rule must be isomorph with the type graph. Also, each graph rule may include a NAC (negative application condition), which indicates that if these conditions were not present in the host graph, then this rule can be applied to the host graph.

There are various tools, including AGG [49], ATOM3 [50], VIATRA2 [51], and GROOVE [25], which are described for the purpose of modeling and analyzing software systems described by the formal modeling language. Each of these tools has a weakness and strength attribute, and each one is used for a particular purpose. Among the aforementioned tools, GROOVE is the only tool that can automatically scan through the production of state space. In addition to this important feature, this tool is open source and new features can be added to it. In this research, we use the GROOVE tool as an environment for implementing and evaluating the proposed techniques, and in the next section, we describe it.

3.3. GROOVE tool

The Groove tool [25] includes a set of tools for modeling the structure of object-oriented systems at runtime, compiling, designing and model transformation is based on the graph transformation which focuses on the use of a MC technique for the identification of object-oriented systems. The major uses of this tool are the presentation of the formalization of model transformation and checking of features to verify the authenticity of software systems [45]. In Groove, a graph is used to describe the system state and graph transforms to express the behavior of the system. The system state space can be constructed using a graph as the initial state and set of rules for the graph transformation.

As an example of a system modeled on this tool, it can refer to the DPP with two philosophers. Fig. 4 defines the host graph which has four nodes: two philosophers and two forks. Each fork is to the left of a philosopher and right of another philosopher. Left and right of the fork are determined by the edge according to Fig. 4 and initial values are also written within each node.

The next step in the groove is the rule definition, LHS, RHS, and NAC graphs are merged and it uses color coding to detect any of its components. For example, thin black nodes and edges belong to both LHS and RHS graphs. The nodes with two edges and blue dot edges belong to the LHS graph (Fig. 5(A)). After exiting the design, values are added from the "edit" plus a "−" sign and blue light to the value of the node (Fig. 5(B)) and after applying the rule on the host graph, they must be removed from the graph. The nodes and edges are green in the RHS graph after applying the rule, they must be created in the host graph, and after removing the design environment, the editing values of the green values plus the "+" sign are added to the node (Fig. 5(A)(B)). Also, nodes with two edges and red dotted edges belong to the NAC graph and their presence in the host graph prevents the application of law [42]. The B part of Fig. 5 shows the "go-hungry" rule. To apply the p: LHS → RHS rule to s mode (which is actually a graph), all LHS graph events are first calculated on s, and then one of them is replaced by the RHS graph. Obviously, matches will be acceptable, including NACs. For example, in accordance, Fig. 5(B), the "go-hungry" rule will be used when the current state of the host graph has a node labeled "think" and by applying this rule on this state, a tag with a tag "think" is deleted and a new edge is created with

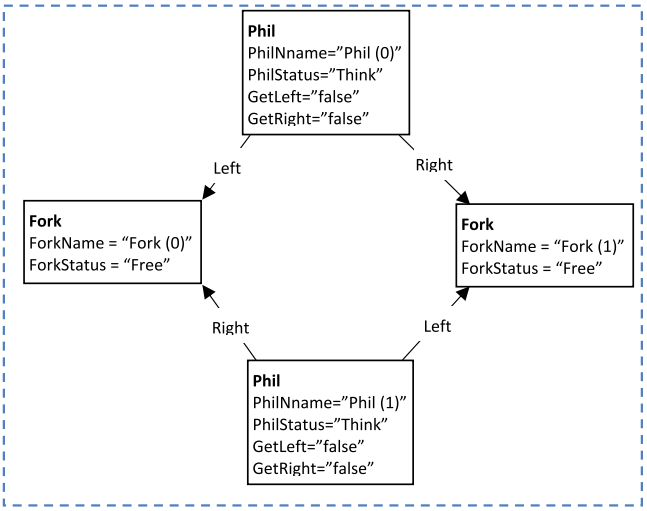

Fig. 4. Host graph on the groove tool for the DPP with two philosophers.

the "hungry" tag (Fig. 5(E)). Fig. 5C, D, E, and F, respectively, show how to implement the relevant rules for "get_left", "get_right", "release_left", and "release_right".

The state space of a model shows its complete behavior, and for its production, all rules must be applied repeatedly to the initial state. Usually, the state space is displayed with a transition system.

Def. 2 (*Transition System*)**.** The transition system T consists of a tuple "⟨S, Act, →, I⟩" in which [52]:

- S: A set of states.
- Act: A set of actions that causes a system state change.
- →: A relation of transition from one state to another via an action, → ⊆S × Act × S.
- I: Shows the initial state.

In the state space of the graph transaction system, we consider the set of transformation rules (R) as the set of acts (Act). For example, Fig. 6 illustrates the state space generated for the DPP with two philosophers. In this state space, the state s12 indicates a target state (in this example, the deadlock of the target. Also, the color path in Fig. 6 shows a witness.

Def. 3 (*Path*)**.** Suppose next (s, r) refers to the state of applying rule r to s state, a path in the state space of a model is defined as the alternating sequence of states and rules applied to them in the form of $s_0 \to r_0 \to s_1 \to r_1 \to s_2 \to \cdots$ for i ≥ 0, and si = next (si-1, ri-1) [45].

The state s_0 is an initial state and the path starting with this state is called the initial path, and the finite paths must end in a finite state. In the example above, s_{12} is a final state. In this context, the sequence path is meant one of the edges and states is initial and final [45]. Like the colored path in Fig. 6. In the models described with the formal language of the graph transformation, for example, in the philosopher's example, at first, only the "go_hungry" rule is valid, and after that, this rule makes the philosopher state the amount of "thinking" to "hungry", which causes The "get_left" rule is also valid, so it can be concluded that the "get_left" rule depends on the "go_hungry" rule. The definition of the dependency between two rules is given below.

Def. 4 (*Rule Dependency*)**.** If r_1 and r_2 are two rules, it is said that the rule r_2 depends on the rule r_1 ($r_1 \to r_2$) if at least one of the following conditions is satisfied [53]:

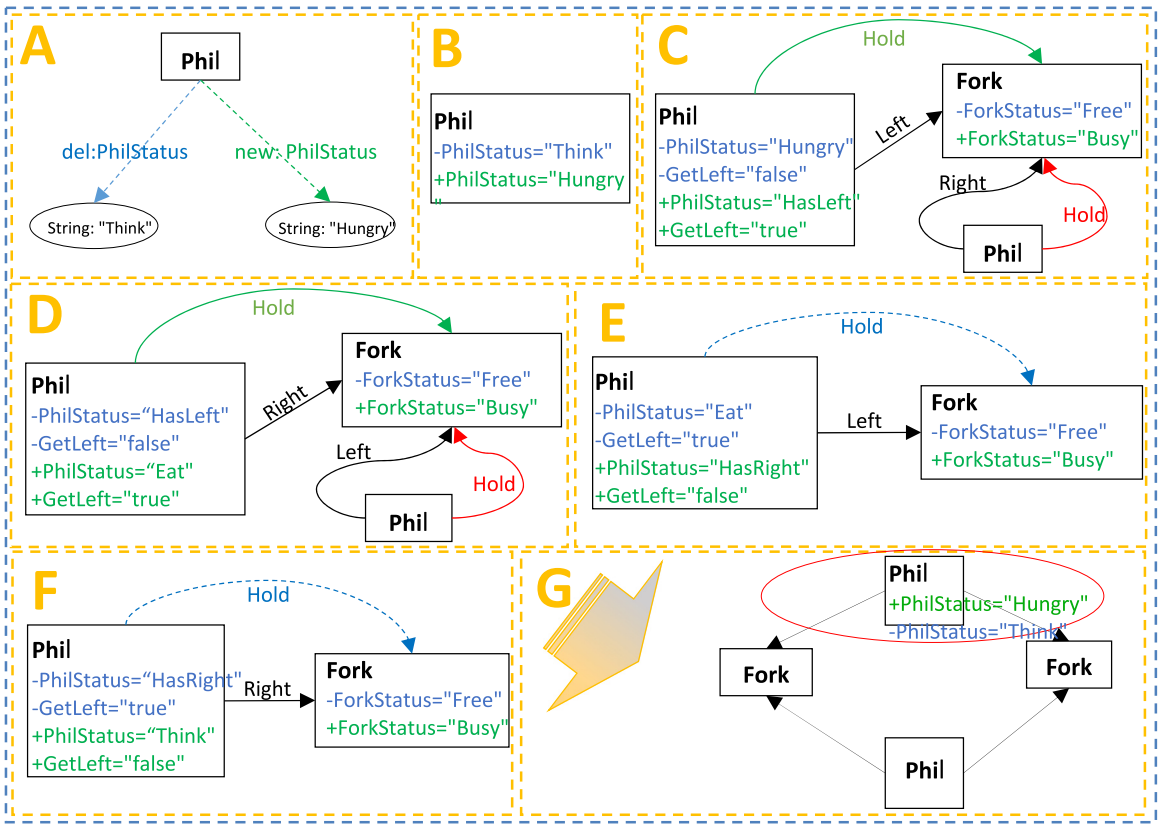

Fig. 5. Details of the designed model for the DPP in the GROOVE Tool.

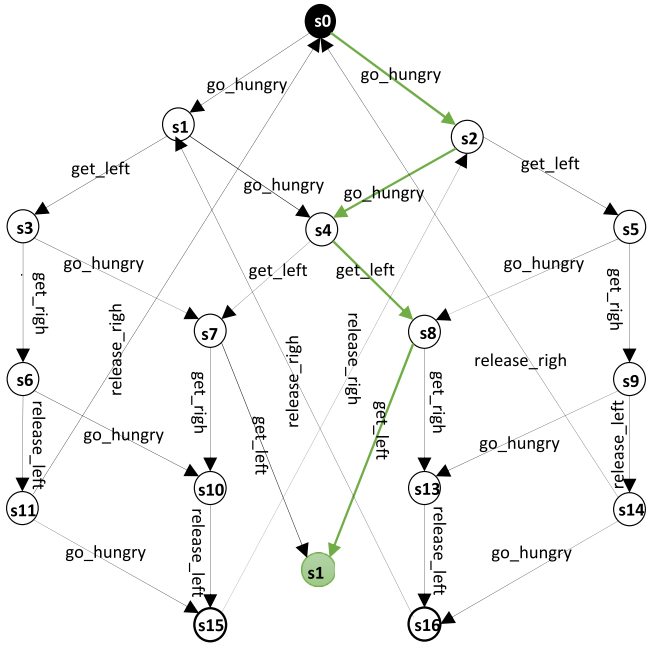

Fig. 6. The state space generated for the DPP with two philosophers.

- At least one edge or node of the LHS graph of the rule r_2 is added to the current state by the RHS graph of the rule r_1.
- At least one edge or node of the NAC graph of the rule r_2 is deleted by the RHS graph of r_1 rule from the current state.

By the other hand, the rule r_2 depends on the rule r_1 if the rule r_1 adds/removes at least one entity (the node or edge) to the current state S (state space) whose existence/loss in the state of next (s, r_1) for applying r_2.

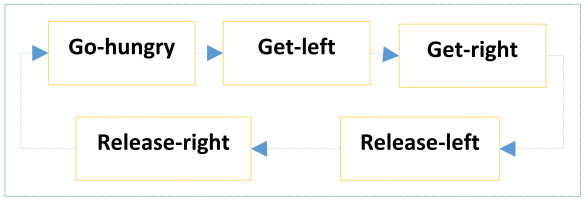

Fig. 7. Rule dependency graph related to the DPP.

Def. 5 (*Rule Dependency Graph*)**.** rule dependency graph for a model is a Directed acyclic graph (DAG) whose nodes and edges indicate the rules of the model and the dependencies between the rules, respectively [44].

For example, Fig. 7 shows the rule dependency graph describing to the DPP.

3.4. Covering Array (CA)

The number of SUT parameters is usually large, and if the values of these parameters are also large, practically it is not possible to check all the states (test cases). A mathematical approach to reducing the number of test cases is CA [6]. In CT, CA is formed in the simplest form of a table. Each row from this table is equivalent to a test case and its columns, the SUT parameters [5] are considered as CA input. In general, CA has four parameters: N, t, p, and v, which are in two forms; (1) CA (N; t, v^p) (if the value of v is constant for all p); and (2) CA (N; t, p, v) is shown [54]. In this regard, p is the number of input parameters; v is

the number of values (or attributes) that each parameter assigns. t is the interaction strength (or t-way interaction) [6]. And N is the number of test cases. In fact, t is a subset of p, which, instead of the p-parameter analysis (column), is considered to be the coverage of the compounds of t from p. This mode is also a Uniform strength CA.

To illustrate CA, we use the DPP example presented Section 3.1. The system consists of five parameters: the philosopher's name (PhilName), fork's number, getting right fork status (GetRight), getting right fork status (GetLeft), philosopher status (PhilStatus), and fork status (ForkStatus). Each of these components is known as an independent parameter and can take one or more values. Table 1 shows these parameters and their values. The tuple (phil (0), fork (0), false, true, thinking, free) forms a test case of the system. To obtain the values of each parameter in this study, the groove tool is used, with the value of the parameters shown in Table 1.

To diagnose an error in a system, we have to produce all the different settings. Each of the settings is considered as a test case, and the total test cases will form the test suite. To accurately identify the error, the test suite should include all possible settings.

In the complete test of the software for the DPP, it is necessary to produce 160 test cases ($2 * 2 * 2 * 2 * 5 * 2 = 160$). Obviously, if parameter numbers and their values are high, this value increases and practically it will be impossible to completely examine the test cases. In addition, since the interaction of the parameter number causes an error, the most intrusive is not the need for a complete review of the test cases. One of the methods for reducing test cases is the CA, which produces the minimum test suite using the t-way method. If p is the number of system parameters, t is in the range of $p \geq t \geq 2$. The value of t defines the t-way component of the parameters. For example, if $t = 2$ (the interaction strength is equal to 2), in the DPP, the ForkName and PhilName parameters must combine the four test modes (string: "Phil (0)", string: "Fork (0)") (string: "Phil (1)"), (string: "Phil (1)", string: "Fork (0)") and (string: "Phil (1) string: "Fork (1)"), and other binary compounds should also have the same conditions as the complete test suite. If t is too high in the test, this leads to the emergence of a large number of a test suite, and if t is low, then the interactions that give rise to the error cannot be identified, so the determination of the exact value of t is very important.

As noted, all the DPP states are 160 test cases. This problem has 6 parameters, of which 5 parameters have 2 values (2^5) and one parameter has 5 value (5^1). Table 2 shows the output of the test suite for the above problem with the interaction strength of 2 (CA (N; 2, 6, $2^5 5^1$). N in the coverage array determines the number of rows in the test suite, and each value is less than this. The test suite production algorithm is more efficient.

In fact, CA is an array N × p whose sample number is N and has the following two characteristics [2]:

- Each column i, $2 \leq i \leq p$, contains members from the set M_i where $v_i = |M_i|$
- The number of rows in each sub-array $N \times t$ of it, all combinations $|M_{pt}| \times \cdots \times |M_{p2}| \times |M_{p1}|$ of t column coverage at least once.

As it was seen, interaction in all parameters was uniform and unchanged, but if a subset of the parameters for the exact error is required to have a higher interaction rate, then the interaction variable is called the VSCA. The array of variables with $VSCA(N, t, p, v\{C\})$ or $VSCA(N, t, v^p, v\{C\})$ is shown. A CA that {C} has one or more $CA(t', p', v)$, and p′ must be a subset of p [3]. It should be noted that in some sources, if the values of the parameters are variable, they are called the Mixed CA (MCA) [3].

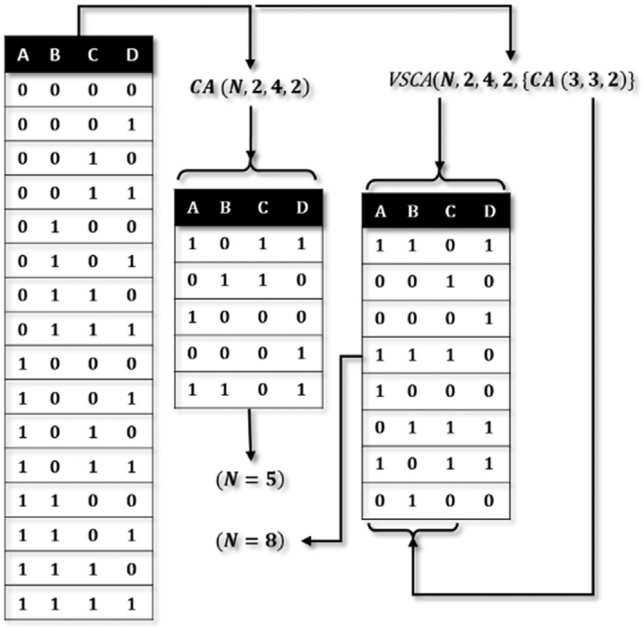

Fig. 8. Illustration of CA and VSCA.

Other constraint array structures include CCAs that have been defined for constraint support. In this structure, a new parameter called forbidden (F) interaction has been added that specifies interactions disallowed in the test suite and displayed as $CA(N; t, v^p, F)$. For example, in CCA (10, 2, 3^3, F), which is F = {(x, 2, 0), (0, x, 1)}, and F tells us that the values (x, 2, 0) and (0, x, 1) for examples of tests is illegal, x is not in the sense meaning and all its elements must be considered [3].

As an example, consider a system with 4 parameters 2 values. In general, the number of test cases for this configuration is $16 = 2^4$ (v^p), which is shown in Fig. 8. But by applying the CA (N; 2, 4, 2) configuration, the number of test samples is reduced to 5. The coverage criterion is that there must be four values (00, 01, 10, and 11) for both columns (AB, AC, AD, BC, BD, CD), and when all possible scenarios are covered, the test suite is completed. In the general case, the total number of coverings obtained for the configurations with parameters equal to the values is given by (1), and the relation (2) for the parameters varies with values.

$$Max\_Coverage = \binom{p}{t} * v^t \tag{1}$$

$$Max\_Coverage = \binom{p}{t} * |v_1| * |v_2| * \cdots * |v_p| \tag{2}$$

For an overview of the VSCA, consider the example VSCA (N; 2, 4, 2 {CA (3, 3, 2)}). In this example, the test case should configure the CA (N1, 2, 4, 2) on the four parameters C, B, A, D and the configuration of CA (N2; 3, 3, 2) on the three parameters A, B, C. Fig. 8 shows its division.

4. The purposed approach

Determining the exact number and values of the parameters manually is difficult, and if the values that the parameters take are dynamic, it is virtually impossible to extract the information manually. One of the most effective ways to overcome this problem is MC. In this article, we try to extract the required primary information from the state space using the GROOVE tool, then obtained information is provided by the metaheuristic algorithms and the CAs are generated among several implementation

Table 1
Output problem values of the DPP by scanning state space.

PhilName	ForkName	GetRight	GetLeft	PhilStatus	ForkStatus
string:"Phil(0)"	string:"Fork(0)"	jbool:false	jbool:false	string:"Think"	string:"Free"
string:"Phil(1)"	string:"Fork(1)"	jbool:true	jbool:true	string:"Hungry"	string:"Busy"
				string:"HasLeft"	
				string:"Eat"	
				string:"HasRight"	

Table 2
The test suite for CA (10; 2, 6, $2^5 5^1$).

PhilName	ForkName	GetRight	GetLeft	PhilStatus	ForkStatus
string:"Phil(0)"	string:"Fork(1)"	jbool:false	jbool:true	string:"Hungry"	string:"Free"
string:"Phil(1)"	string:"Fork(0)"	jbool:false	jbool:false	string:"HasRight"	string:"Busy"
string:"Phil(0)"	string:"Fork(0)"	jbool:true	jbool:false	string:"Think"	string:"Free"
string:"Phil(1)"	string:"Fork(1)"	jbool:true	jbool:true	string:"HasLeft"	string:"Busy"
string:"Phil(0)"	string:"Fork(1)"	jbool:true	jbool:false	string:"Eat"	string:"Busy"
string:"Phil(1)"	string:"Fork(0)"	jbool:false	jbool:true	string:"Eat"	string:"Free"
string:"Phil(0)"	string:"Fork(1)"	jbool:true	jbool:true	string:"HasRight"	string:"Free"
string:"Phil(1)"	string:"Fork(0)"	jbool:true	jbool:false	string:"Hungry"	string:"Busy"
string:"Phil(1)"	string:"Fork(1)"	jbool:false	jbool:true	string:"Think"	string:"Busy"
string:"Phil(0)"	string:"Fork(0)"	jbool:false	jbool:false	string:"HasLeft"	string:"Free"

algorithms (GA, PSO, BAT, and TLBO). Our GA model produces far better results than the existing approaches, and then we can minimize CA by using a new approach. A noticeable point can be made after the production of the test suite, which can be used to decide on the test suite produced by MC, which is discussed in this article. To illustrate the optimality of the proposed approach, we selected three most commonly algorithms that used in the field of CA, namely BA, PSO and TLBO. We also generate the test suite in the GROOVE tool with these three algorithms (Fig. 9(A)), then we can make an acceptable evaluation because there is no strategy (A strategy to incorporate the model into the model checking tool to produce the CA as input) in this field to compare with the GA. Also for more evaluation, we implement the proposed approach out of the GROOVE environment so that we can perform the evaluation with available strategies (Fig. 9(B)). In the next section, we describe the CA manufacturing process with the GROOVE tool.

4.1. Generating the state space (Step 1)

To generate CA, we first recommend a SUT model based on the topics discussed in Section 3.3. After modeling the system, we generate the model state space using the search algorithms implemented in GROOVE (Fig. 6).

4.2. Extracting the parameters and their values (Step 2)

In this step, each single created state should be checked and new values added to the parameters. The first mode is s_0, which represents the system's initial state, In this case, you can extract the initial values of the variables defined in the host graph. The parameters and values obtained in the initial state are shown in Fig. 10. After s_0, two states s_1 and s_2 are executed with the "go_hungry" rule (Fig. 10(A)), which is executed by two philosophers "Phil (0)" and "Phil (1)". The implementation of this rule makes the situation of the philosopher from the state of "think" to the state of "hungry". As a result, the "hungry" amount is added to the "PhilStatus" parameter, as in Fig. 10(B). The s_4 mode occurs when one of the philosophers already has a "hungry" state, so this does not add a bit to the parameters. States with numbers 3, 5, 7, and 8 are triggered when a philosopher intends to get the left fork and consequently, the "HasLeft", "Busy" and "true" values are added to the parameters of "PhilStatus", "ForkStatus" and "GetLeft" respectively (Fig. 10(C)). States 6, 9, 10, and 13 occur when the philosophers hold the left fork and they want to get the right fork, in which case the philosopher will go to "Eat" (Fig. 10(D)) and "true" are added to the "GetRight" parameter and "Eat" to the "PhilStatus" parameter. After eating, the philosopher first releases the left fork, "PhilStatus" goes to "HasRight", and this value is also added to the "PhilStatus" parameter, which is in the states 11, 14, 15, and 16 in accordance with Fig. 10(E). The state 12 also does not add any values to the parameters (Fig. 10(F)).

4.3. Generating Covering Matrix (CM) (Step 3)

One of the most time-consuming functions in CA production with AI algorithms is to calculate the weight of a test case. To calculate the weight of a test case, which is described below, there is a need for a large search across a large data structure. In this paper, we use the data structure in [3], which considers a cell for each test case. This data structure has C(p, t) rows, and each column and number of possible states for that composition is retained. To understand more about the data structure, the DPP with $t = 2$ is the structure of Fig. 11(A). As you can see, the first row represents the combination of the first parameter (PhilName) and the second parameter (ForkName), where the number one in the first column and the number two in the second column represent these two parameters at the beginning of the row, and then go to the number $|\text{PhilName}| * |\text{ForkName}|$. Zero is in the next cell. The following lines are created the same way. For the calculation of weight from the CM, consider the sample test (Phil (0), Fork (1), False, True, HasRight, and Free). For convenience in calculations, we use numeric values instead of string values. As a result, the test case is converted to (0, 1, 0, 1, 4, and 0). Now, to calculate the weight of this test case, it is sufficient to scan the row to the CM line. Given that t = 2, we extract the values of the first and the second columns of each row of test case values. For example, the fourth line that shows the combination of 1 and 5, the value (0, *, *, *, 4, *) is extracted from the test sample (Fig. 11(B)). Then we calculate the decimal value of this value (0, 4) and the equivalent column of this value is checked. If it is zero, it means not covering the test case, then add it and add one unit to the test suite (Fig. 11(C)). If it is one, it means that the test case has already been covered and the weight remains unchanged.

4.4. GA implementation (Step 4)

To achieve the optimal test case, we implemented and evaluated various algorithms. That is, the BAT, PSO, TLBO, and GA

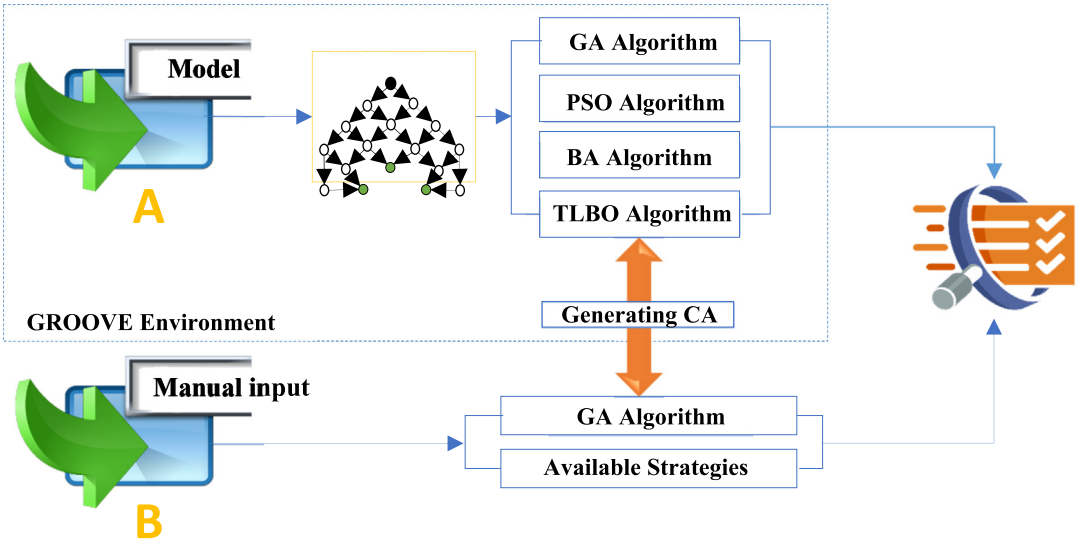

Fig. 9. Types of evaluation in the proposed approach.

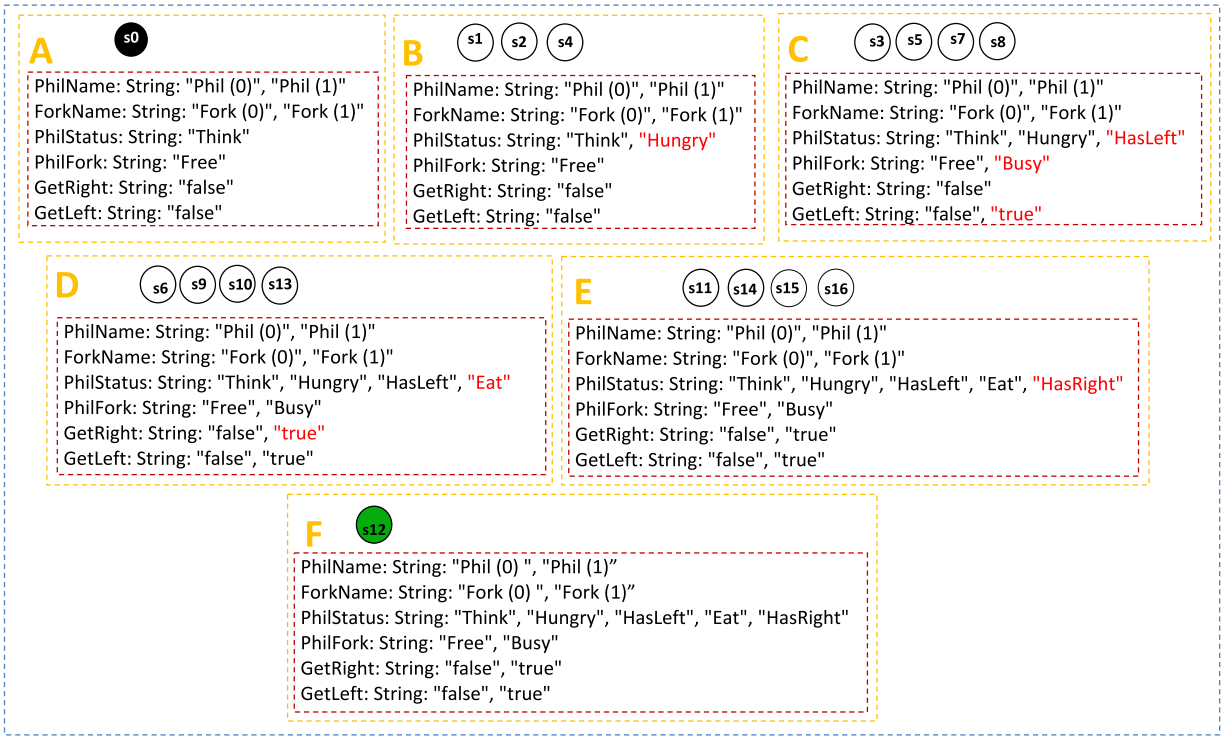

Fig. 10. Extracting the parameter values by traversing the state space.

algorithms produce better results than their competitors, and previous studies also confirm the correctness of this fact. Among these four algorithms, the GA produced far better results. The GA used in this study uses the structure used in [3], with the exception that after applying the crossover and mutation function, the test case is compared with a randomized test case. The routine of other implementation algorithms has been unchanged, so refraining from redefining their steps in this section is avoided. As mentioned, the GA begins with chromosome encoding. Since the values of the parameters are often stringy and the thread management is difficult in the calculations, we use numeric values instead of the string. In the proposed method, the values of each chromosome gene are numerically between 0 and v − 1 and v is the number of values that one parameter can receive. So it can be concluded that chromosome encoding uses value encoding in this article. Each chromosome consists of p gene and its value ranges from zero to v − 1. Fig. 12 shows a chromosome for the DPP. This chromosome is equivalent to the test case (Phil (0), Fork (1), true, true, Eat, Busy). For other algorithms implemented in this paper, the same method is used to initialize particles and the initial population.

The next step in AI algorithms is to calculate the test case weight, which is described in the second step of this section. The crossover function is one of the most important functions in producing the optimal test suite. First, the best test case is selected from the population, and we call it best, then it sends the best to the crossover function. Finally, two random values

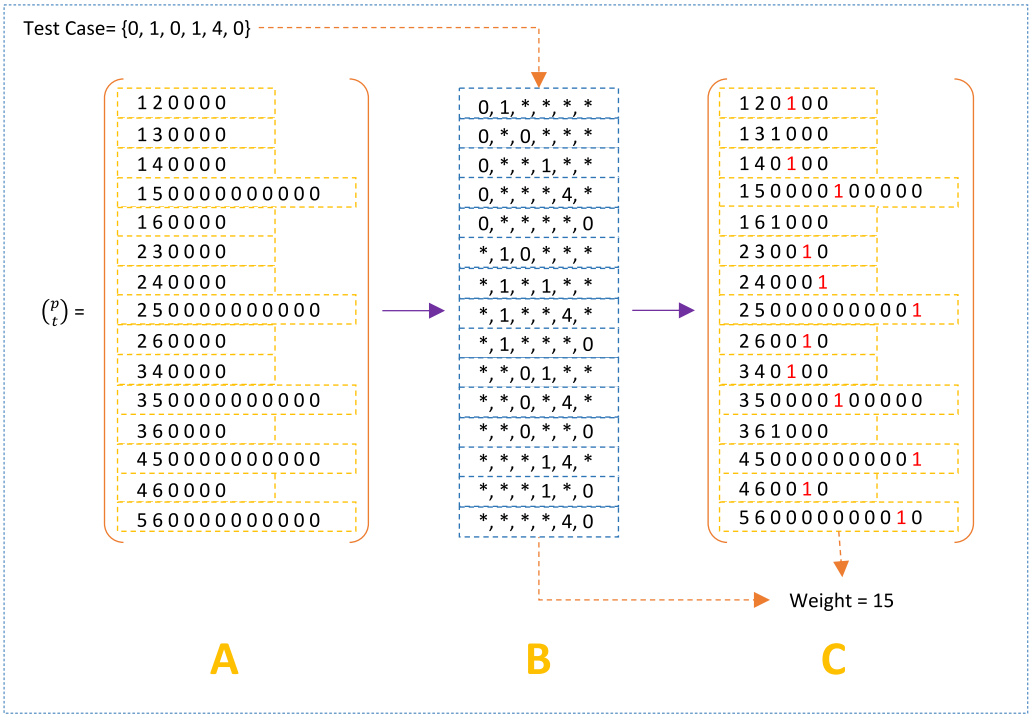

Fig. 11. CM generation and calculation of test case weight.

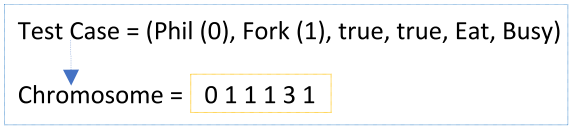

Fig. 12. A chromosome for the DPP.

```
OffSpring[0][i]=(int)(rndi*best[i]+(rndj)*best[i-1])
OffSpring[1][i]=(int)((1-rndi)*best[i]+((1-rndj)*best[i-1])
```

Fig. 13. Crossover function in the proposed approach.

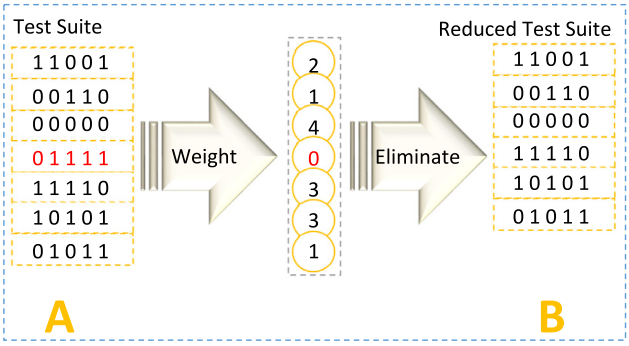

Fig. 14. Elimination of redundant test cases.

between zero and one of the chromosome genes are calculated as Fig. 13.

The mutation function used in this article uses the uniform method introduced in [3]. In this way, three of the best genes are randomly selected. Their random values are replaced by them. One of the major problems in the production of the test suite is the local optima. In this solution, for the purpose of avoiding the local optima after the production of each child a random test case is also produced. If the value of this test case is better than the test case generated by mutation and crossover functions, the random value is replaced with the best value.

4.5. Test suite minimization (Step 5)

As we know, AI-based strategies use the OTAT method. This means that at each step, a test case with the highest weight is added to the test suite. It may also cover the states covered by this test case. Therefore, this test case is actually used, which should be removed from the final test suite. With an example of the steps to minimize the test suite, consider the CA (N; 2, 5, 2) configuration. The maximum possible weight for a test case is $C(p,t) = 10$. At this stage, the weight of each test case is recalculated, but for weight purposes here, the number of states is covered only by the relevant test case. For example, consider the first test case in the test suite of Fig. 14(A). In columns 1 and 2, the value (1, 1, *, *, *) occurred both in the first test case and in the fifth test case. Therefore, this amount is not unique, but the amount (1, *, 0, *, *), as well as (*, 1, *, 0, *) only occur in the first test case, and the other cases in this test case are in the other test cases, hence the weight of this test case is 2 and cannot be removed. Consider the case of the fourth row. All quantities of this test case have taken place in other test cases. So this case should be removed from the test suite. As a result, the number of levels in this test suite decreases to 6 (Fig. 14(B)).

The general stages of implementing the proposed strategy are presented in Fig. 15.

4.6. Setting parameters

Parameters are very important in the metaheuristic algorithms. In CA, the population size parameter (PopSize) is much more important than other parameters. Given the wide variety of configurations, larger configurations require a larger search

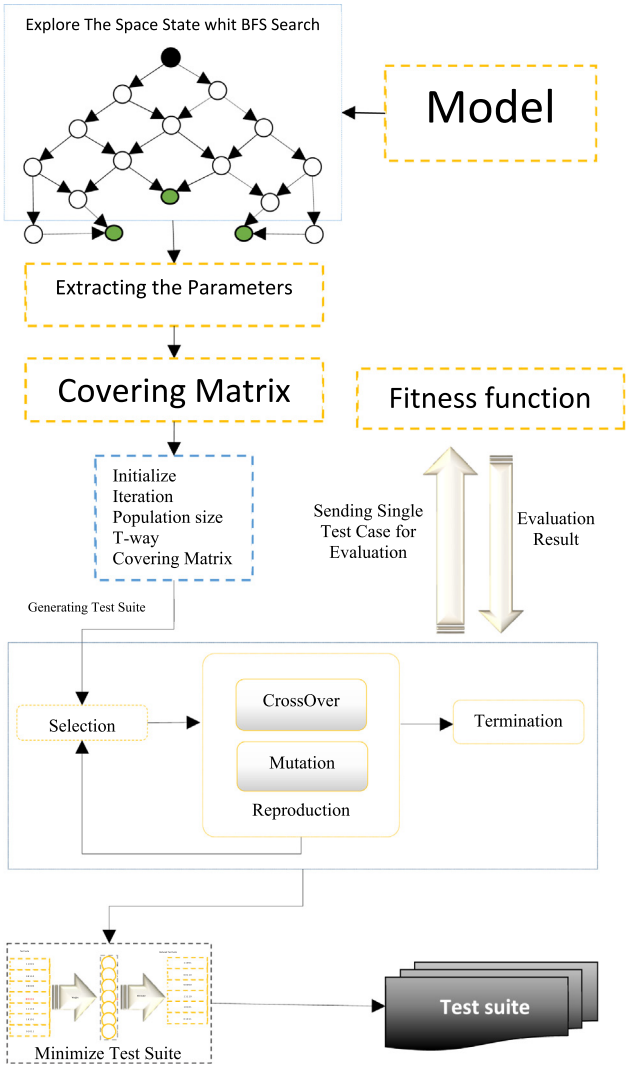

Fig. 15. The flowchart of the proposed approach.

space. Therefore, a larger population is needed to find the optimal solution. In this study, we consider variable parameters for the population in 4 algorithms TLBO, BAT, PSO and GA, and other parameters are constant. To investigate the effect of changing parameters on the size of the test suite, the four configurations CA (N; 2, 6, 4), CA (N; 2, 7, 5), CA (N; 2, $4^3 5^3 6^2$) and CA (N; 4, 9, 3) are considered. For the PSO algorithm, the value of C1 = 2, C2 = 3 and W = 0.8, the BAT algorithm is A = 0.5 and B = 0.5, Crossover and mutation rate are 0.8 and eventually Repetition = 20. And because of the avoidance of an unreasonable increase in the volume of the paper, the effect of changing these values on the test suite size has been discarded and only the population size is evaluated.

In the configuration of CA (N; 2, 6, 4), we examine the population between 5 and 90 individuals (particles). The PSO algorithm has the best performance in this configuration with the test suite of 21 test cases and it gets this value in PopSize = 50, so it can be concluded that this configuration does not require increasing the population to get the best value, but as shown in Fig. 16(A), the average array size is 100 repetitive to PopSize = 90. The BAT algorithm also has the same results as the PSO algorithm, with the difference that the array is the lowest for the test suite of 22 test cases.

The TLBO algorithm in PopSize = 60 gets the test suite with the size of 22 test case but works in a much weaker way than the previous two algorithms. The GA also has PopSize = 30 and the test suite has 22 test cases as the best extraction. But it is better than other patterns in terms of average.

It can be concluded that in the configuration of CA (N; 2, 6, 4), the best value for PopSize for GA, BAT, PSO and TLBO is 30, 50, 50, and 60 individuals (particles), respectively (Fig. 16(A)). In the configuration of CA (N; 2, 7, 5), the best values produced for GA, BAT, PSO, and TLBO are 35, 37, 37 and 38 test cases available in PopSize with values of 40, 70, 70 and 70. Therefore, these values are the most appropriate choice for population size. In this configuration, the GA has the best performance for the average array size, and in Fig. 16(B) the corresponding values are shown. The best value for configuring CA (N; 2, $4^3 5^3 6^2$) in GA, PSO, BAT, and TLBO is 31, 41, 41, and 43, respectively. The GA with a population of 50 is the best. But other algorithms require more populations to reach to the best. BAT and TLBO algorithms with 100 individuals and PSO with 110 can produce the best values. Fig. 16(C) shows changes in population growth over the mean size of the test suite related to the CA configuration (N; 2, $4^3 5^3 6^2$). Fig. 16(D) is devoted to the population size survey for the configuration of CA (N; 4, 9, 3). The best values by GA, BAT, PSO, and TLBO algorithms are 189, 192, 192, and 195, respectively, in GA with 90, and other algorithms with a population of 150 individuals capable of generating these values.

5. Evaluation

Evaluation in the production of CA is divided into two major categories of the array size and the production time of the test suite. The array size is independent of the hardware and software platform and depends only on the implementation of the algorithm, but the runtime depends on the hardware and the operating system [5]. The evaluation of time and array size in this study is divided into two parts. The first part, considering that there is no strategy to get the model as input, we tried to implement a number of metaheuristic algorithms for this purpose. Since the number of meta-heuristic algorithms is too high, and implementing all of them is not practically feasible, so in this project, three algorithms TLBO, BA and PSO, which previously were used in coverage array production, are selected and implemented. The results of which are compared with our genetic algorithm. Also, to demonstrate the proper power of our proposed algorithm, we introduce the GA algorithm out of the GROOVE environment and compare it with other existing strategies. In this section, we try to make comparisons with a limited number of strong strategies. This comparison is just to show that the proposed solution is appropriate for the time and array size. The specifications for the hardware platform for implementing the proposed method are Windows 7, 2.20 GHz core™ i7QM CPU and 6GB RAM. The coding is open in Java (JDK 1.8) and in the eclipse environment. In the evaluation tables, NA (Not Available) means non-publication in articles, and NS (Not Supported) means does not support the strategy of the corresponding configuration. The best results in each row are highlighted. In each evaluation, the castle is represented by two values: best and average. These two values are obtained from 100 repetitions for some configurations. Table 3 shows the values for the parameters BA, TLBO, PSO, and GA. The code of our strategy is publically available on the web.[1]

[1] https://www.dropbox.com/s/izf2ha5887b292y/CoveringArray.rar?dl=0.

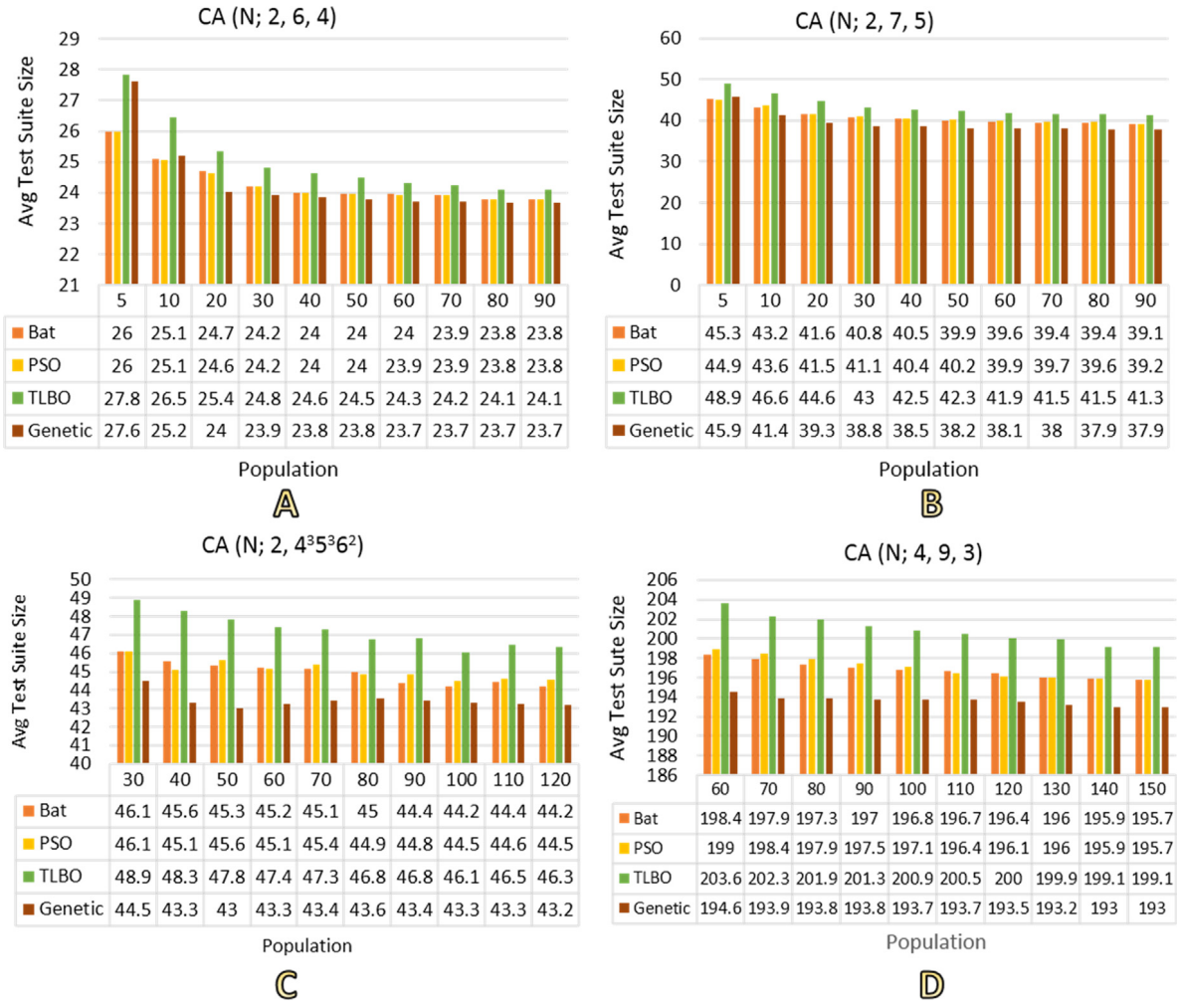

Fig. 16. Effect of increasing PopSize on the average size of the test suite.

Table 3
Parameter values for metaheuristic algorithms.

Algorithm	Parameters	Value
GA	Population	10–100
	Crossover rate	40%–10%
	Mutation rate	40%–100%
	Selection method	Tournament selection
	Crossover method	Arithmetic
	Mutation method	Uniform
PSO	Population	50–150
	Repetition	20–50
	W	0.8
	C1	2
	C2	3
BA	Population	50–150
	Repetition	20–50
	Min frequency	0
	Max frequency	100
	Loudness	0.25
	Pulse rate	0.5
TLBO	Population	50–150
	Repetition	20–50

5.1. Evaluation with implemented strategies

For evaluation in this section, we will examine 5 known case studies we consider the first case study sample as the DPP, which was thoroughly investigated. The four following items are the Online Shopping System (OSS) [55], Bug Tracking System (BTS) [56], Travel Agency System (TAS) [57] and an example of hotel management [52,55,56]. We give a brief explanation of each of these.

OSS is a process of purchasing products through the Internet so that customers can view the products of the company and then order a list of your required items after payment through a credit card after the payment is made by mail to the customer's address. The customer can also cancel the order before returning the order to the credit card [55].

BTS is used in software development projects to facilitate the management of reported software bugs. Among the basic features of these systems is the management of several projects, the user and the relationships between them. In addition, users can assign different roles (manager, developer, and testator). Each role is allowed to perform certain activities in the system. The developer team as well as software testers, with the help of this system, can interact with each other properly for reporting, prioritizing and correcting the error. As the first case study, we tested the function of a sample of software error tracking systems and used its graph conversion descriptions to generate a test case [56].

In TAS, they also offer services such as airline ticket reservations, hotel reservations, and so on. And the Hotel Example [52] has similar conditions with the travel agency.

5.1.1. DPP evaluation

The first evaluation of this paper is shown in Table 4. In this table, the eating philosophers for interaction between 2 and 4 are

examined. A major parameter in this problem is the number of philosophers. The more the number of philosophers is, the state space created by the MC increases and the values of the dynamic variables also change. In this evaluation, we will examine 2 to 5 philosophers. As you can see, in this evaluation, the two PSO and GA solutions have better results than BA and TLBO solutions. At t = 2, the four strategies produce approximately equal outcomes, and only when the number of philosophers is 5, GA has better results than other algorithms. At t = 3, both GA and PSO strategies outperform their rivals, and of course GA is better than the PSO. At t = 4 the PSO performs better than the others. The TLBO and BA strategies have very close results, but TLBO has a mild advantage over BA.

5.1.2. OSS evaluation

OSS is an example to justify this research. In this system, the number of customers and the type of purchase are changing dynamically over time. And whatever goes forward, the parameter values also change. So, in this section, we examine the number of scanned states for OSS evaluation. The higher the value is, the greater the value of the variables is, because the number of online customers is changing dynamically. The configuration of this master case is very large. Table 5 is dedicated to the evaluation of this system. In this evaluation, we consider the value of t as in the previous section i.e. 2, 3, and 4. In this evaluation, the GA strategy produces very good results compared to the three TLBO, BA, and PSO strategies.

5.1.3. BST evaluation

In this problem, the configuration is much lighter than OSS. As you can see, Table 6 is considered for the evaluation of BTS in this evaluation. All four GA, TLBO, BA, and PSO strategies produce acceptable results, and in many cases the best results are extracted by each of the four strategies. But in general, the GA strategy is more powerful than other strategies.

5.1.4. HE evaluation

In this evaluation, as in the previous two evaluations, with an increasing number of states, the value of the variables increases. The value of t in this section is also 2, 3, and 4, which evaluate 1000, 2000, 4000, and 8000 states. In this case, the increase in the number of states from 1000 to 2000 and from 2000 to 4000 does not change the number and amount of parameters. In this evaluation, GA strategy is more powerful than other strategies. And after that, the PSO produces better results, and the two BA and TLBO strategies produce similar results, but TLBO has a relative advantage (Table 7).

5.1.5. TAS evaluation

One of the most heavily studied case studies in CA is TAS. This configuration has a total of 25 parameters, which is very large and it calculates the CA for that time. Therefore, for saving time, only 2 and 3 are considered for t. In these configurations, the GA strategy is also stronger. And Table 8 contains the corresponding values, and the average is only 5 repetitions.

5.2. Evaluation with the existing strategies

To compare the suggested genetic algorithm, we implement this solution outside of GROOVE and, as usual, the CA inputs are entered manually. The proposed solution has favorable results in terms of time and size of the array. And in terms of time, it is faster than all AI-based strategies, and in terms of size, DPSO's strategy has the best performance among the strategies. It should be noted that some of the existing strategies in [13] may have even better results even with DPSO. In this evaluation, we compare the results with two strategies based on the PSO, DPSO, and PSTG algorithms, as well as pure computational strategies such as IPOG, TConfig, PICT and Jenny. The comparisons of this section with the four tables are completed. The table separation criterion is placed on the value t, as Table 9 provides an evaluation for the configurations with the interaction strength 2, Table 10, the configurations with the interaction strength 3, Table 11, configurations with interaction 4 and finally Table 12 evaluates interaction strengths more than 4.

The first comparison in this section is assigned to t = 2 and the evaluation results are shown in Table 9. As you can see, in this evaluation, AI-based strategies are much more powerful than computational strategies. Among the strategies based on computing, the IPOG-D and IPOG strategy provides more relevant results for each other, and then the PICT, TConfig and Jenny strategies extract better results, respectively. As you can see, PSTG strategy in 5 configurations, DPSO strategy in 8 cases, CS strategy in 8 cases and GA strategy in 9 cases extract the best results.

The next comparison is assigned to t = 3, and the results are presented in Table 10. It is noted that the higher the interaction is, the more discriminating AI-based strategies. In this assessment, for the CA (N; 3, 3^4) configuration, three AI-based strategies with the production of a test suite with a size of 27 test samples have the best performance. In CA (N; 3, 2^7) and CA (N; 3, 3^5) GA strategy, in CA (N; 3, 3^6) DPSO strategy and in other configurations, both GA and DPSO strategies have the best performance.

Another comparison of this section is in Table 11 which contains the results of the assessment case t = 4. This assessment also confirms the superiority of AI-based strategies on computational strategies. In this assessment, two GA and DPSO strategies produce better results In CA (N; 4, 2^7), CA (N; 4, 2^{10}) and CA (N; 4, 3^6), the GA strategy and CA (N; 4, 3^5) and CA (N; 4, 3^7) DPSO strategies in other cases, the results of these two strategies are equal and better than other strategies.

For t > 4, as seen in Table 12, some "NS" and "> day" values are considered for some of the strategies. NS (Not Supported) means that the strategy is not supported by that configuration and "day" also refers to the lack of production of a test suite in less than one day. The IPOG-D, IPOG, PSTG and CS strategies for t > 6 are in NS mode. TConfig does not have the ability to generate the test suite for the configurations in Table 12 in less than one day. Needless to say, we have implemented these strategies on our system and it is possible that in a stronger system, this strategy can produce a test suite for less than one day. The Jenny, PICT, and DPSO strategies can support t up to 15, 16, and 9, respectively, to produce a test suite at one time, but the proposed strategy has the ability to extract the test suite up to t = 20. The configurations used in this section are very heavy, and so their production time is very high. Therefore, the results observed in Table 12 are the result of the repetition of 2 or 3 times of implementation for artificial intelligence strategies. The PopSize in some of these configurations has also dropped to 20. In terms of the array size, the GA strategy is very suitable and has an acceptable superiority to other strategies.

We continue the evaluation to compare time. The PopSize, Repetition, CrossRate, and MutatRate parameters are values that affect the speed of the algorithm execution. For this comparison, we consider 50, 50, 0.4 and 0.4 for the above parameters, respectively. In this section, to show the speed of each algorithm, consider the time taken to generate a test sample. For example, suppose that the DPSO algorithm succeeds in producing the test suite with 14 samples for 3 s for the CA (N; 2, 3^7) configuration. As a result, the production time of each test sample is 214 (14/3 $*$ 1000) ms in average.

In order to compare the time of strategies, it is necessary to implement them on the same system. For time evaluation,

Table 4
Evaluation results of CA generation for the DPP.

t	#Phil	BA N.Best	BA N.AVG	PSO N.Best	PSO N.AVG	TLBO N.Best	TLBO N.AVG	**GA** N.Best	**GA** N.AVG
2	2	10	10.2	10	10.2	10	10.3	10	10.0
	3	15	16.2	15	16.0	15	16.1	15	15.8
	4	21	22.7	20	22.5	21	22.7	20	22.2
	5	26	29.5	26	28.5	26	29.1	26	28.6
3	2	30	31.3	30	31.2	30	31.3	30	30.7
	3	48	49.3	47	49.4	48	49.0	45	48.5
	4	82	83.4	81	83.2	82	83.2	81	83.3
	5	128	131.2	126	129.9	128	131.4	127	129.6
4	2	55	59.2	58	60.8	55	58.7	55	58.3
	3	112	115.7	110	114.5	111	115.6	113	115.5
	4	178	180.3	176	180.1	178	180.3	177	180.9
	5	296	271.7	265	270.3	267	271.5	264	269.5

Table 5
Evaluation results of CA generation for the OSS.

t	#state	BA N.Best	BA N.AVG	PSO N.Best	PSO N.AVG	TLBO N.Best	TLBO N.AVG	**GA** N.Best	**GA** N.AVG
2	1000	66	67.6	66	67.5	62	65.1	62	64.5
	2000	78	80.9	79	81.9	79	80.2	77	78.2
	4000	98	101.0	97	100.7	98	100.8	91	94.7
	8000	127	130.3	127	130.5	127	130.2	118	123.5
3	1000	277	479.2	476	478.9	470	473.8	466	472.1
	2000	627	633.3	625	632.2	625	632.6	609	614.2
	4000	868	870.6	865	869.7	866	869.9	829	837.0
	8000	1307	1320.4	1309	1319.4	1311	1318.4	1246	1253.5
4	1000	3160	3162.2	3155	3160.2	3158	3161.4	3071	3077.8
	2000	4469	4475.6	4468	4475.3	4468	4476.7	4259	4274.6
	4000	6880	6887.2	6880	6889.9	6880	6890.2	6571	6578.2
	8000	11441	11447.1	11447	11457.7	11447	11457.7	11033	11040.2

three artificial intelligence algorithms and five computational-based algorithms are used. One of the few available strategies is the DPSO which is used in this evaluation. For both CS and PSO strategies, the algorithms implemented in [5] are utilized. Among the computation-based strategies, 5 powerful strategies are used, namely Jenny, TConfig, PICT, IPOG-D and IPOG. Their evaluation results are shown in Tables 13–15. Table 13 shows the impact of increasing t on time. As it can be seen, both IPOG and IPOG-D strategies need near-zero time to produce a case. In terms of time, these two strategies are the most powerful CA production strategies. Jenny and PICT are also much more powerful than artificial intelligence-based strategies. But TConfig strategy is even weaker than artificial intelligence-based strategies in terms of time. Among the AI strategies, the GA is much stronger than its competitors. Also, the results of this strategy are better than TConfig, PICT and Jenny. Tables 14 and 15 also show the impact of increasing p and v on the production time of a test case in CA, respectively. As in the previous evaluation, both IPOG and IPOG-D strategies are much stronger than their competitors.

5.3. Statistical test

Using the Wilcoxon signed rank sum test is very common for evaluation [58] . This test takes into account the magnitude of the difference between the pairs. In other words, this test is used to examine the semantic difference between the two sets of numbers. The SPSS tool can be used to use this test. For both strategies A and B, ranks show the number of samples A $>$ B, A $<$ B and A $=$ B, and the Test Statistics section contains two z and Asymp values. Sig (2-tailed) that the description of z is not included in this article and the other determines the meaningful difference that if this value is less than 0.05 it means that there is a significant difference between the two strategies. In Table 16, which shows the output of the Wilcoxon test on the results of Tables 4–8, each part is considered for comparing a table. The first part is related to Table 4, which, as shown, does not have a meaning difference with other algorithms, as Wilcoxon defines. But by referring to the Ranks column, we find that GA is superior to BA and PSO and is equal to TLBO in every respect.

Table 6
Evaluation results of CA generation for the BST.

t	#State	BA N.Best	BA N.AVG	PSO N.Best	PSO N.AVG	TLBO N.Best	TLBO N.AVG	**GA** N.Best	**GA** N.AVG
2	1000	25	27.2	25	27.1	25	27.0	25	26.9
	2000	31	33.8	31	33.7	31	33.8	31	33.8
	4000	48	49.4	48	49.2	48	49.3	48	49.1
	8000	64	65.1	63	64.4	64	64.9	63	64.6
3	1000	102	107.6	102	107.1	102	107.2	101	107.7
	2000	154	162.7	154	162.2	153	162.3	153	162.8
	4000	246	256.6	246	255.6	246	256.0	246	255.5
	8000	320	328.9	319	328.3	319	328.5	323	329.9
4	1000	501	508.9	499	508.3	500	508.5	492	503.4
	2000	622	631.2	620	630.6	620	630.5	620	629.2
	4000	741	748.0	740	747.0	740	747.3	736	748.3
	8000	966	979.3	964	975.7	965	976.2	963	975.2

Table 7
Evaluation results of CA generation for the HE.

t	#State	BA N.Best	BA N.AVG	PSO N.Best	PSO N.AVG	TLBO N.Best	TLBO N.AVG	**GA** N.Best	**GA** N.AVG
2	1000	25	26.6	25	26.4	25	26.5	24	26.6
	2000	25	26.6	25	26.4	25	26.5	24	26.6
	4000	29	31.8	29	31.6	29	31.8	29	31.3
	8000	29	31.8	29	31.6	29	31.8	29	31.3
3	1000	110	113.9	109	113.4	109	113.7	107	112.4
	2000	110	113.9	109	113.4	109	113.7	107	112.4
	4000	148	152.3	146	151.6	147	151.8	145	150.2
	8000	148	152.3	146	151.6	147	151.8	145	150.2
4	1000	416	422.1	414	421.5	414	422.0	413	419.3
	2000	416	422.1	414	421.5	414	422.0	413	419.3
	4000	648	650.4	644	649.0	645	649.6	641	645.2
	8000	648	650.4	644	649.0	645	649.6	641	645.2

Table 8
Evaluation results of CA generation for the TAS.

t	#State	BA N.Best	BA N.AVG	PSO N.Best	PSO N.AVG	TLBO N.Best	TLBO N.AVG	**GA** N.Best	**GA** N.AVG
2	1000	42	43.3	42	43.3	41	43.3	41	43.3
	2000	61	64.7	60	64.2	60	64.5	60	62.9
	4000	61	64.7	60	64.2	60	64.5	60	62.9
	8000	72	74.8	72	74.4	72	74.5	71	73.7
3	1000	228	230.3	228	230.2	228	230.2	227	228.3
	2000	428	430.5	424	429.0	426	429.1	400	402.5
	4000	428	430.5	424	429.0	426	429.1	400	402.5
	8000	560	571.2	569	573.5	560	570.2	526	230.5

Table 9
Comparison of proposed solution array size with other strategies at t = 2.

	Pure computation strategies					AI-based strategies				
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSTG	DPSO	**GA**	**GA**
	N	N	N	N	N	N.Best	N.Best	N.Best	N.Best	N.AVG
CA (N; 2, 2^7)	8	7	7	8	7	6	6	7	6	6.2
CA (N; 2, 3^3)	9	10	10	15	9	9	9	9	9	9.7
CA (N; 2, 3^4)	13	10	13	15	9	9	9	9	9	9.9
CA (N; 2, 3^5)	14	14	13	15	15	11	12	11	11	12.0
CA (N; 2, 3^6)	15	15	14	15	15	13	13	14	13	14.2
CA (N; 2, 3^7)	16	15	16	15	15	14	15	14	14	14.7
CA (N; 2, 3^8)	17	17	16	15	15	15	15	15	15	16.1
CA (N; 2, 3^9)	18	17	17	15	15	15	17	15	15	16.2
CA (N; 2, 3^{10})	19	17	18	21	15	16	17	16	16	17.2
CA (N; 2, 4^7)	28	28	27	32	29	25	26	24	24	25.4
CA (N; 2, 5^7)	37	40	40	45	45	37	37	34	36	37.7

Table 10
Comparison of proposed solution array size with other strategies at t = 3.

	Pure computation strategies					AI-based strategies				
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSTG	DPSO	**GA**	**GA**
	N	N	N	N	N	N.Best	N.Best	N.Best	N.Best	N.AVG
CA (N; 3, 2^7)	14	16	15	14	16	12	13	15	12	12.5
CA (N; 3, 3^4)	34	32	34	27	32	28	27	27	27	27.9
CA (N; 3, 3^5)	40	40	43	45	41	38	39	41	37	38.2
CA (N; 3, 2^{10})	18	20	18	18	20	17	17	16	16	16.6
CA (N; 3, 3^6)	51	48	48	45	46	43	45	33	37	38.7
CA (N; 3, 3^7)	51	55	51	50	55	48	50	48	48	49.0
CA (N; 3, 3^8)	58	58	59	50	56	53	54	52	52	53.8
CA (N; 3, 3^9)	62	64	63	71	63	58	58	56	56	57.8
CA (N; 3, 3^{10})	65	68	65	71	66	62	62	59	59	61.9
CA (N; 3, 3^{11})	65	72	70	76	70	66	64	63	63	65.5

Table 11
Comparison of proposed solution array size with other strategies at t = 4.

	Pure computation strategies					AI-based strategies				
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSTG	DPSO	**GA**	**GA**
	N	N	N	N	N	N.Best	N.Best	N.Best	N.Best	N.AVG
CA (N; 4, 2^7)	31	36	32	40	35	27	29	34	27	27.7
CA (N; 4, 2^{10})	39	45	43	51	46	28	34	34	25	25.4
CA (N; 4, 3^5)	109	97	100	162	97	94	96	81	88	89.5
CA (N; 4, 3^6)	140	141	142	162	141	132	133	131	129	132.3
CA (N; 4, 3^7)	169	166	168	226	167	154	155	150	152	154.8
CA (N; 4, 3^8)	187	190	189	226	192	173	175	171	171	173.6
CA (N; 4, 3^9)	206	213	211	260	210	195	195	187	187	190.4
CA (N; 4, 3^{10})	221	235	231	278	233	211	210	206	206	209.6

The second part is related to Table 5, which shows that the GA model is superior in this comparison to its competitors, and this has a meaning difference with other algorithms. The third part is related to Table 6. In this comparison, the GA algorithm with BA algorithm has a semantic difference, but with the two TLBO and PSO algorithms there is no meaning difference, but the Ranks column proves the superiority of GA. The fourth and fifth parts correspond respectively to Tables 7 and 8. The results show that in this GA comparison, with other strategies, there is a meaning difference.

Another assessment of the Wilcoxon test is related to Tables 9–15, which is assigned to this assessment in Table 17. Table 17 consists of four parts. As can be seen, in the first, second and third parts of Table 17, which are respectively Tables 9–11. GA, except for DPSO, has a meaningful difference with other algorithms and its superiority is quite tangible. To compare the exact GA with DPSO, you can use the Ranks section, from which

Table 12
Comparison of proposed solution array size with other strategies at t > 4.

	Pure computation strategies					AI-based strategies				
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSTG	DPSO	**GA**	**GA**
	N	N	N	N	N	N.Best	N.Best	N.Best	N.Best	N.AVG
CA (N; 5, 3^7)	458	477	452	678	466	434	441	438	431	438.5
CA (N; 6, 3^8)	1466	1515	1455	1493	1409	1399	1401	1409	1398	1410.0
CA (N; 7, 3^9)	4746	>day	4618	NS	NS	NS	NS	4451	4437	4453.5
CA (N; 8, 3^{10})	14999	>day	14599	NS	NS	NS	NS	13933	13907	13943.0
CA (N; 9, 3^{11})	47009	>day	45521	NS	NS	NS	NS	>day	43808	43880.0
CA (N; 10, 3^{12})	147004	>day	141990	NS	NS	NS	NS	>day	136096	136109.5
CA (N; 11, 3^{12})	305797	>day	278993	NS	NS	NS	NS	>day	267630	2677690.0
CA (N; 12, 2^{14})	9422	>day	9112	NS	NS	NS	NS	8972	8890	8912.5
CA (N; 13, 2^{14})	13251	>day	12441	NS	NS	NS	NS	>day	10251	10283.5
CA (N; 14, 2^{15})	26579	>day	25036	NS	NS	NS	NS	>day	23377	23399.0
CA (N; 15, 2^{16})	53977	>day	51127	NS	NS	NS	NS	>day	46575	46598.5
CA (N; 16, 2^{17})	>day	>day	100266	NS	NS	NS	NS	>day	95675	95684.0
CA (N; 17, 2^{18})	>day	>day	>day	NS	NS	NS	NS	>day	179540	179610.5
CA (N; 18, 2^{19})	>day	>day	>day	NS	NS	NS	NS	>day	330391	330408.0
CA (N; 19, 2^{20})	>day	>day	>day	NS	NS	NS	NS	>day	624940	624987.0
CA (N; 20, 2^{20})	>day	>day	>day	NS	NS	NS	NS	>day	1048576	1048698.5

Table 13
Comparison of the impact of increasing t on the production time of a test case.

	Pure computation strategies					AI-based strategies			
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS [3]	PSO	DPSO	**GA**
	N	N	N	N	N	time	time	time	time
CA (2, 3^7)	2.50	5.33	0.63	0.06	0.07	18.67	106.67	214	1.41
CA (3, 3^7)	1.76	7.82	0.78	0.02	0.02	62.00	90.00	479	2.21
CA (4, 3^7)	1.79	49.64	0.54	0.00	0.01	39.74	105.03	1040	2.29
CA (5, 3^7)	1.57	152.96	1.55	0.09	0.00	43.81	79.73	436	1.25
CA (6, 3^7)	1.01	462.02	1.18	0.05	0.00	31.86	99.90	202	0.97

Table 14
Comparison of the impact of increasing p on the production time of a test case.

	Pure computation strategies					AI-based strategies			
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSO	DPSO	**GA**
	N	N	N	N	N	time	time	time	time
CA (3, 3^4)	0.29	2.19	1.18	0.04	0.03	48.15	150.00	74	0.36
CA (3, 3^5)	0.75	2.50	2.09	0.02	0.02	55.00	156.41	171	0.73
CA (3, 3^6)	1.76	6.46	2.71	0.02	0.02	58.44	166.67	303	1.67
CA (3, 3^7)	1.37	7.82	4.51	0.02	0.02	62.40	164.00	479	2.84
CA (3, 3^8)	1.21	21.96	6.10	0.02	0.02	73.45	172.22	692	3.54
CA (3, 3^9)	1.29	26.88	9.05	0.01	0.02	78.17	181.03	982	5.19
CA (3, 3^{10})	1.54	41.76	9.85	0.01	0.02	87.50	182.26	1373	7.76
CA (3, 3^{11})	1.85	56.14	10.00	0.01	0.01	107.88	200.00	1825	9.67
CA (3, 3^{12})	2.65	74.86	10.97	0.01	0.01	120.14	202.99	2415	12.23

Table 15
Comparison of the impact of increasing v on the production time of a test case.

	Pure computation strategies					AI-based strategies			
	Jenny	TConfig	PICT	IPOG-D	IPOG	CS	PSO	DPSO	**GA**
	N	N	N	N	N	time	time	time	time
CA (3, 2^7)	2.86	1.88	6.00	0.07	0.06	63.08	141.67	400.00	2.01
CA (3, 3^7)	1.76	7.82	3.73	0.02	0.02	42.00	164.00	500.00	2.31
CA (3, 4^7)	0.97	22.95	4.27	0.01	0.01	55.46	183.90	482.14	2.09
CA (3, 5^7)	2.58	13.31	3.24	0.00	0.00	61.80	191.15	796.30	2.24
CA (3, 6^7)	2.50	31.09	2.42	0.00	0.00	45.16	210.95	515.07	2.32

Table 16
Wilcoxon signed rank sum test for Tables 4–8.

		Ranks			Test statistics	
		GA<	GA>	GA=	Z	Asymp. Sig. (2-tailed)
Table 4	BA-GA	6	1	5	−1.930	.054
	TLBO-GA	3	3	6	−.213	.832
	PSO-GA	6	1	5	−1.552	.121
Table 5	BA-GA	11	1	0	−2.353	.019
	TLBO-GA	12	0	0	−3.059	.002
	PSO-GA	11	0	1	−2.934	.003
Table 6	BA-GA	7	1	4	−1.761	.078
	TLBO-GA	5	1	6	−1.276	.202
	PSO-GA	5	1	6	−1.265	.206
Table 7	BA-GA	10	0	2	−2.873	.004
	TLBO-GA	10	0	2	−2.873	.004
	PSO-GA	10	0	2	−2.842	.004
Table 8	BA-GA	8	0	0	−2.588	.010
	TLBO-GA	6	0	2	−2.232	.026
	PSO-GA	5	0	3	−2.041	.041

Table 17
Wilcoxon signed rank sum test for Tables 9–15.

		Ranks			Test statistics	
		GA<	GA>	GA=	Z	Asymp. Sig. (2-tailed)
Table 9	Jenny-GA	10	0	1	−2.831	.005
	TConfig-GA	11	0	0	−2.971	.003
	PICT-GA	11	0	0	−2.965	.003
	IPOG-D-GA	9	0	2	−2.670	.008
	IPOG-GA	6	1	4	−2.043	.041
	CS-GA	2	0	9	−1.414	.157
	PSTG-GA	6	0	5	−2.278	.023
	DPSO-GA	2	1	8	−0.000	1.00
Table 10	Jenny-GA	10	0	0	−2.819	.005
	TConfig-GA	10	0	0	−2.806	.005
	PICT-GA	10	0	0	−2.825	.005
	IPOG-D-GA	8	1	1	−2.392	.017
	IPOG-GA	10	0	0	−2.840	.005
	CS-GA	8	0	2	−2.555	.011
	PSTG-GA	1	0	1	−2.840	.005
	DPSO-GA	2	2	7	−.184	.854
Table 11	Jenny-GA	8	0	0	−2.521	.012
	TConfig-GA	8	0	0	−2.524	.012
	PICT-GA	8	0	0	−2.524	.012
	IPOG-D-GA	8	0	0	−2.524	.012
	IPOG-GA	8	0	0	−2.524	.012
	CS-GA	7	0	1	−2.374	.018
	PSTG-GA	8	0	0	−2.536	.011
	DPSO-GA	3	2	3	−.680	.496
Table 12	Jenny-GA	11	0	0	−2.934	.003
	TConfig-GA	2	0	0	−1.342	.180
	PICT-GA	12	0	0	−3.059	.002
	IPOG-D-GA	2	0	0	−1.352	.180
	IPOG-GA	2	0	0	−1.352	.180
	CS-GA	2	0	0	−1.342	.180
	PSTG-GA	2	0	0	−1.342	.180
	DPSO-GA	5	0	0	−2.023	.043
Table 13	Jenny-GA	3	2	0	−0.134	.892
	TConfig-GA	5	0	0	−2.022	.043
	PICT-GA	2	3	0	−1.213	.224
	IPOG-D-GA	0	5	0	−2.022	.043
	IPOG-GA	0	5	0	−2.022	.043
	CS-GA	5	0	0	−2.022	.043
	PSO-GA	5	0	0	−2.022	.043
	DPSO-GA	5	0	0	−2.022	.043
Table 14	Jenny-GA	2	7	0	−2.191	.028
	TConfig-GA	9	0	0	−2.665	.008
	PICT-GA	8	1	0	−2.191	.028
	IPOG-D-GA	0	9	0	−2.665	.008
	IPOG-GA	0	9	0	−2.665	.008
	CS-GA	9	0	0	−2.665	.008
	PSO-GA	9	0	0	−2.665	.008
	DPSO-GA	9	0	0	−2.665	.008
Table 15	Jenny-GA	3	2	0	−0.134	.892
	TConfig-GA	4	1	0	−1.752	.080
	PICT-GA	5	0	0	−2.022	.043
	IPOG-D-GA	0	5	0	−2.022	.043
	IPOG-GA	0	5	0	−2.022	.043
	CS-GA	5	0	0	−2.022	.043
	PSO-GA	5	0	0	−2.022	.043
	DPSO-GA	5	0	0	−2.022	.043

it can be concluded that GA is superior to DPSO. The fourth part is also in Table 12. Considering that many results are not available for strategies, and these values are considered as misses in the Wilcoxon test, and this test ignores somewhat the corresponding values of the missing value. This has a negative effect on the final result. However, GA has a different meaning in the three strategies of Jenny, PICT, and DPSO, and there is no meaning difference with the three strategies of TConfig, IPOG, and PSTG, which is why the three strategies are only two possible Production of the test suite. The remainder of the evaluation in Table 17 also relates to time (Tables 13–15). As can be seen, OPAT strategies are much faster than their competitors. In general, artificial intelligence-based strategies are very slow in producing coverage array. The evaluation results show that the proposed solution is much faster than artificial intelligence-based solutions and even some computational solutions.

6. Conclusion and future work

CA is an important branch of CT. A lot of research has been done on CA production, which mostly attempts to reduce the size of the test suite or reduce production time, and have been able to some extent eliminate the concern to produce the minimum test suite at the appropriate speed. In this research, we tried to highlight the new CA challenge and we managed it using MC. The challenge is that the manual input in previous strategies, and the error rate was very high. But in this study, by using the GROOVE tool, we could reduce the percentage of data entry errors and provide CA with this powerful MC tool. In this research, we extracted the parameters and the corresponding values using four GA, BA, PSO, and TLBO algorithms. The results of the evaluation showed that GA is much stronger than the other three algorithms, which is due to a change in the crossover and mutation function. Also, to minimize the CA, a simple minimum size function is used which is very effective. To prove that GA has appropriate results in terms of time and array size, this approach compares with several strong strategy examples, including DPSO, PSTG, CS, Jenny, TConfig, IPOG and IPOG-D, and the results of the evaluation showed that they were very appropriate in terms of time and array size. But as mentioned, this research is not aimed solely at the goal of producing CAs at the proper time and can have various applications.

The first use of MC is to extract “basic information” for CA, which is very important, but MC can play a very significant role in CA. Sometimes the error does not occur just in a test case. In other words, error detection requires a suite of the test cases from the test suite. For example, in the DPP, the deadlock can be an error. We know that the occurrence of a deadlock requires the scrolling of the suite from the test cases. Therefore, the order of the test cases in the test suite is important. CA uses SCA to generate the suite of test cases. But how will this suite be selected to detect the deadlock? One of the common methods is to examine all possible modes. And the best tool for choosing this sequence can be MC. The second application and significance of this research can be the production of this test suite. For example, consider the same

DPP. Consequently, SCA production can be considered with a MC as the next research.

The main problem of the MC is the state space explosion, which is, of course, the problem of this research. As noted above, to obtain initial CA information, the state space was first created using BFS or DFS. This would cause the state space explosion for complex systems. Is it possible to extract all the parameters and their values using the heuristic methods without having to create the entire state space completely? Or can you avoid creating insignificant states to prevent a space state explosion? The quest for answers to these questions can be a major future study in MC and CA.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.asoc.2020.106219.

CRediT authorship contribution statement

Sajad Esfandyari: Conceptualization, Methodology, Software, Writing - original draft. **Vahid Rafe:** Supervision, Writing - review & editing.

References

[1] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, L. Zhang, TCA: An efficient two-mode Meta-Heuristic Algorithm for combinatorial test generation (T), in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, 9-13 Nov. 2015.

[2] H. Wu, C. Nie, F.-C. Kuo, H. Leung, C.J. Colbourn, A discrete particle Swarm Optimization for covering array generation, IEEE Trans. Evol. Comput. 19 (4) (2015) 575–591.

[3] S. Esfandyari, V. Rafe, A tuned version of genetic algorithm for efficient test suite generation in interactive t-way testing strategy, Inf. Softw. Technol. 94 (2018) 165–185.

[4] A.R.A. Alsewari, K.Z. Zamli, Design and implementation of a harmony-search-based variable-strengtht-way testing strategy with constraints support, Inf. Softw. Technol. 54 (6) (2012) 553–568.

[5] B.S. Ahmed, T.Sh. Abdulsamad, M.Y. Potrus, Achievement of minimized combinatorial test suite for configuration-aware software functional testing using the Cuckoo Search algorithm, Inf. Softw. Technol. 66 (2015) 13–29.

[6] B.S. Ahmed, K.Z. Zamli, C. PengLim, Application of Particle Swarm Optimization to uniform and variable strength covering array construction, Appl. Soft Comput. 12 (4) (2012) 1330–1347.

[7] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, The AETG system: an approach to testing based on combinatorial design, IEEE Trans. Softw. Eng. 23 (7) (1997) 437–444.

[8] A.W. Williams, Determination of test configurations for Pair-Wise interaction coverage, in: IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems: Tools and Techniques, Kluwer, B.V., Deventer, The Netherlands, 2000, pp. 59–74.

[9] B. Jenkins, Jenny download web page, Bob Jenkins' Website, 2019, [Online]. Available: http://burtleburtle.net/bob/math/jenny.html.

[10] A. Hartman, IBM intelligent test Case Handler, IBM alphaworks, 2019, [Online]. Available: http://www.alphaworks.ibm.com/tech/whitch.

[11] J. Czerwonka, Pairwise testing in real world: practical extensions to test case generator, in: 24th Pacific Northwest Software Quality Conference, IEEE Computer Society, Portland, OR, USA, 2006, pp. 419–430.

[12] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: a general strategy for t-way software testing, in: 4th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, IEEE Computer Society, Tucson, AZ, 2007.

[13] C.J. Colbourn, Covering array tables for t=2, 3, 4, 5, 6, 2019, [Online]. Available: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html.

[14] M. Grochtmann, K. Grimm, Classification trees for partition testing, Softw. Test. Verif. Reliab. 3 (2) (1993) 63–82.

[15] M.N. Borazjany, L.Sh. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, An input space modeling methodology for combinatorial testing, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013.

[16] P. Satish, A. Paul, K. Rangarajan, Extracting the combinatorial test parameters and values from UML sequence diagrams, in: IEEE International Conference on Software Testing, Verification, and Validation Workshops, Cleveland, OH, USA, 2014.

[17] P. Satish, K. Sheeba, K. Rangarajan, Deriving combinatorial test design model from UML activity diagram, in: IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Luxembourg, 2013.

[18] C.T. Hu, D.R. Kuhn, T. Xie, J. Hwang, Model checking for verification of mandatory access control models and properties, Int. J. Softw. Eng. Knowl. Eng. 21 (1) (2011) 103–127.

[19] R.C. Bryce, Y. Lei, D.R. Kuhn, R. Kuhn, Combinatorial testing, in: Software Engineering and Productivity Technologies, 2010, pp. 196–207.

[20] R. Kuhn, Y. Lei, R. Kacker, Practical combinatorial testing: Beyond pairwise, IT Prof. 10 (3) (2008) 19–23.

[21] L. Kampel, B. Garn, D.E.S.E. Simos, Combinatorial methods for modelling composed software systems, in: 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, Tokyo, Japan, 2017.

[22] L. Sh. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, T. Xie, D. Kung, A combinatorial testing-based approach to fault localization, IEEE Trans. Softw. Eng. (2018) http://dx.doi.org/10.1109/TSE.2018.2865935, (Accepted/in press).

[23] M. Grindal, J. Offutt, Input parameter modeling for combination strategies, in: IASTED International Conference on Software Engineering (SE2007), Feb 13-15, Innsbruck, Austria, 2007.

[24] M. Grindal, Handling Combinatorial Explosion in Software Testing (Doctoral thesis), in: monograph (Other academic), Linköping University, Department of Computer and Information Science. Linköping University, 2007.

[25] H. Kastenberg, A. Rensink, Model checking dynamic states in GROOVE, in: International SPIN Workshop on Model Checking of Software, Vol. 3925, 2006, pp. 299–305.

[26] T. Shiba, T. Tsuchiya, T. Kikuno, Using artificial life techniques to generate test cases for combinatorial testing, in: 28th Annual International Computer Software and Applications Conference, Hong Kong, China, 2004.

[27] A. Hartman, Graph theory, Combinatorics and algorithms, in: Software and Hardware Testing using Combinatorial Covering Suites, Vol. 34, Springer, US, 2005, pp. 237–266.

[28] Y.-W. Tung, W. Aldiwan, Automating test case generation for the new generation mission software system, in: 2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484), Big Sky, MT, USA, USA, 2000.

[29] K.Z. Zamli, M.F.J. Klaib, M.I. Younis, N.A.M. Isa, R. Abdullah, Design and implementation of a t-way test data generation strategy with automated execution tool support, Inform. Sci. 181 (9) (2011) 1741–1758.

[30] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing, Softw. Test. Softw. Test. Verif. Reliab. 18 (3) (2008) 125–148.

[31] J. Stardom, Metaheuristics and the Search for Covering and Packing Array (Thesis (M.Sc.)), Simon Fraser University, 2001, p. 2001.

[32] M.B. Cohen, Designing Test Suites for Software Interactions Testing (PHD Thesis), University of Auckland, Department of Computer Science, Auckland, 2004.

[33] T. Mahmoud, B.S. Ahmed, An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use, 42 (22) (2015) 8753-8765.

[34] K.Z. Zamli, B.S. Ahmed, T. Mahmoud, W. Afzal, in: Y. Tan (Ed.), Fuzzy Adaptive Tuning of a Particle Swarm Optimization Algorithm for Variable-Strength Combinatorial Test Suite Generation, in: Swarm Intelligence, vol. 3, IET, 2018.

[35] K.Z. Zamli, F. Din, S. Baharom, 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. 59 (2017) 35–50.

[36] F. Din, K.Z. Zamli, Pairwise test suite generation using adaptive teaching learning-based optimization algorithm with remedial operator, in: International Conference of Reliable Information and Communication Technology, 2018.

[37] A.B. Nasser, K.Z. Zamli, A new variable strength t-way strategy based on the Cuckoo Search Algorithm, in: Intelligent and Interactive Computing, Springer, Singapore, 2019, pp. 193–203.

[38] A.B. Nasser, K.Z. Zamli, B.S. Ahmed, Dynamic solution probability acceptance within the flower pollination algorithm for combinatorial t-way test suite generation, in: Intelligent and Interactive Computing, Springer, Singapore, 2019, pp. 3–11.

[39] A.B. Nasser, K.Z. Zamli, A.A. Alsewari, B.S. Ahmed, Hybrid flower pollination algorithm strategies for t-way test suite generation, PLOS ONE 13 (5) (2018).

[40] Y.A. Alsariera, M.A. Majid, K.Z. Zamli, A Bat-inspired strategy for pairwise testing, ARPN J. Eng. Appl. Sci. 10 (2015) 8500–8506.

[41] Y.A. Alsariera, A.B. Nasser, K.Z. Zamli, Benchmarking of Bat-inspired interaction testing strategy, Int. J. Comput. Sci. Inform. Eng. (IJCSIE) 7 (2016) 71–79.

[42] K.Z. Zamli, B.Y. Alkazemi, G. Kendall, A Tabu Search hyper-heuristic strategy for t-way test suite generation, 44 (2016) 57–74.
[43] E. Pira, V. Rafe, A. Nikanjam, EMCDM: Efficient model checking by data mining for verification of complex software systems specified through architectural styles, Appl. Soft Comput. 49 (2016) 1185–1201.
[44] E. Pira, V. Rafe, A. Nikanjam, Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm, J. Syst. Softw. 131 (2017) 181–200.
[45] E. Pira, V. Rafe, A. Nikanjam, Searching for violation of safety and liveness properties using knowledge discovery in complex systems specified through graph transformations, Inf. Softw. Technol. 97 (2018) 110–134.
[46] V. Rafe, M. Darghayedi, E. Pira, MS-ACO: a multi-stage ant colony optimization to refute complex software systems specified through graph transformation, Soft Comput. (2018) 1–26.
[47] X. He, Z. Ma, W. Shao, G. Li, A metamodel for the notation of graphical modeling languages, in: 31st Annual International Computer Software and Applications Conference, Beijing, China, 2007.
[48] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT press, 2006.
[49] G. Taentzer, AGG: A graph transformation environment for modeling and validation of software, in: International Workshop on Applications of Graph Transformations with Industrial Relevance, Berlin, Germany, 2003, 446–453.
[50] J. de Lara, H. Vangheluwe, ATOM3: A tool for multi-formalism and metamodeling, in: International Conference on Fundamental Approaches to Software Engineering, Vol. 2306, 2002, pp. 174–188.
[51] D. Varró, A. Balogh, The model transformation language of the VIATRA2 framework, Sci. Comput. Program. 68 (3) (2007) 214–234.
[52] A. Kalaee, V. Rafe, Model-based test suite generation for graph transformation system using model simulation and search-based techniques, Inf. Softw. Technol. 108 (2019) 1–29.
[53] T. Mens, R.V.D. Straeten, M. D'Hondt, Detecting and resolving model inconsistencies using transformation dependency analysis, in: 9th International Conference, Model Driven Engineering Languages and Systems, Genova, Italy, 2006.
[54] K.Z. Zamli, F. Din, G. Kendall, B.S. Ahmed, An experimental study of Hyper-Heuristic selection and acceptance mechanism for combinatorial t-way test suite generation, Inform. Sci. 399 (2017) 121–153.
[55] G. Engels, B. Güldali, M. Lohmann, Towards model-driven unit testing, in: International Conference on Model Driven Engineering Languages and Systems Models in Software Engineering, 2006.
[56] O. Runge, T.A. Khan, R. Heckel, Test case generation using visual contracts, in: Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013), Vol. 58, 2013.
[57] V. Rafe, Scenario-driven analysis of systems specified through graph transformations, 24 (2013) 136–145.
[58] S. García, D. Molina, M. Lozano, F. Herrera, A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization, J. Heuristics 15 (6) (2008).