DivSampCA: A Tuple-oriented Adaptive Sampling for Generating Small Pairwise Covering Arrays

Anonymous Authors

***Abstract*—Combinatorial interaction testing (CIT) is an effective method for verifying highly configurable systems, with pairwise testing being the most commonly used CIT technique due to its strong defect detection capabilities. A major challenge in pairwise testing is the pairwise covering array generation (PCAG) problem, which aims to cover all valid pairwise combinations with the minimal number of configurations. Existing greedy PCAG methods typically draw configuration samples of excessive size because they do not adequately focus on directly optimizing pairwise tuple coverage when generating configurations, and they fail to exploit opportunities to use a single configuration to cover multiple tuples simultaneously. To address these limitations, we propose *DivSampCA*. It employs a tuple-oriented adaptive sampling technique that dynamically refines sampling probabilities to enhance the tuple-covering capacity of the generated configurations. Moreover, it utilizes a full coverage strategy to maximize the number of tuples covered by a single configuration, ensuring that all valid pairwise combinations are covered with as few configurations as possible. We validated *DivSampCA* on 124 publicly available PCAG benchmark instances, demonstrating that our approach generates covering arrays approximately 25% smaller than those produced by the state-of-the-art PCAG algorithms. These results indicate that *DivSampCA*, which effectively reduces the size of covering arrays while maintaining full coverage, is a significant advancement in sampling techniques for solving the PCAG problem.**

***Index Terms*—Pairwise Testing, Covering Array, Adaptive Sampling**

I. Introduction

With the rapid advancement of technology and the growing diversity of business requirements, software systems increasingly demand a high degree of configurability to adapt to various application scenarios [1], [2]. A highly configurable system is characterized by numerous configuration options, enabling users to tailor different objects of the program to meet specific requirements [3]. Customers can assign different values to these options to suit their individual needs.

In practical scenarios, testing all possible configurations of a configurable system to identify interaction errors between different options is inefficient or even impractical. For small-scale systems with a small number of configuration options, exhaustive testing may still be feasible. However, as the number of configuration options increases, the potential configurations grow exponentially. For example, a system with 30 options, each having 2 possible values, would require $2^{30} = 1,073,741,824$ tests, which is practically impossible.

To address this challenge, *combinatorial interaction testing* (CIT) systematically samples a representative subset of configurations from the configuration space to construct a covering array [4]. It guarantees that, for a specified strength t, every valid t-wise combination of option values is contained in at least one sampled configuration, thereby ensuring the detection of failures triggered by t-way interactions. Considering that achieving full coverage takes a long time for large values of t, *pairwise testing* (CIT with $t = 2$) is commonly adopted in practice. Pairwise testing aims to generate a *pairwise covering array* (PCA) that includes all possible value pairs. Extensive empirical studies on real-world configurable systems have shown that most defects can be detected through pairwise testing [5]–[7], highlighting its efficacy as a testing methodology.

Existing CIT methods can be broadly classified into exact algorithms (*e.g.*, [8]–[13]), greedy strategies (*e.g.*, [14]–[22]), and meta-heuristic approaches (*e.g.*, [23]–[29]); all three exhibit efficiency bottlenecks when applied to large-scale problem instances. Exact algorithms compute optimal PCAs via exhaustive search; however, their prohibitive computational complexity restricts applicability to problem instances of small size. Greedy algorithms construct PCAs by iteratively selecting configurations that locally maximize the number of newly covered feature interactions; although this yields PCAs rapidly, the strategy cannot guarantee full coverage of all t-way interactions even after extensive sampling [30]. Meta-heuristic algorithms are presently able to generate comparatively small PCAs, but their time-intensive global exploration and local refinement render the approach prohibitively expensive for large-scale systems [31].

Recent advances in formal reasoning tools have catalyzed efficient constructions of PCAs. The prevailing greedy approaches integrate high-performance SAT or SMT solvers to sample a large pool of diversified configurations, from which a compact subset is extracted as the configuration sample [32]–[35]. These methods employ metrics such as Hamming distance or chi-square distance as quantitative criteria to measure divergence between configurations during sampling. However, such metrics do not directly indicate whether the pairwise tuples entailed by the observed differences remain uncovered, which in turn limits their utility for increasing tuple coverage. Moreover, once the majority of combinations have been

covered, the remaining tuples are extremely sparse within the configuration space, consequently, continued massive sampling not only consumes excessive computational resources but also fails to achieve full coverage.

To address these limitations, we propose a two-phase adaptive sampling algorithm named *DivSampCA*. During the sampling phase, *DivSampCA* introduces a new technique called *tuple-oriented adaptive sampling (TOAS)*. The algorithm iteratively updates the sampling probability of each literal in proportion to its frequency within the uncovered tuples, thereby fostering the generation of configurations toward those expected to cover a greater number of uncovered tuples. During the full coverage phase, *DivSampCA* incorporates an innovative strategy dubbed *more tuples per case (MTPC)*, which constructs configurations capable of covering multiple uncovered tuples simultaneously. By leveraging the synergistic interaction between these two core mechanisms, *DivSampCA* effectively addresses the limitations of existing greedy methods, providing a more efficient solution to the *pairwise covering array generation* (PCAG) problem.

We conduct extensive experiments to evaluate the performance of *DivSampCA*. Specifically, we used 124 publicly available instances modeled from real-world configurable systems as benchmarks. *DivSampCA* was compared with state-of-the-art PCAG algorithms, including *SamplingCA* [36], *CAmpactor* [37] and *LS-Sampling-Plus* [38]. The experimental results show that *DivSampCA* can not only generate smaller PCAs more quickly than existing full coverage algorithms but also achieve higher pairwise coverage compared with high t-coverage algorithms. These results demonstrate that *DivSampCA*, through the use of *TOAS* and *MTPC*, represents a significant advancement in addressing the PCAG problem.

In summary, the contributions of this paper are as follows:

- We propose a tuple-oriented adaptive sampling method, termed *TOAS*, which enhances diversity of the sampled configurations.
- We develop an efficient full covering strategy called *MTPC*, designed to minimize the number of configurations required to cover all remaining uncovered tuples.
- We conduct extensive experiments to evaluate the performance of *DivSampCA*. The results show that *DivSampCA* effectively generates smaller covering arrays, marking a notable improvement in PCAG algorithm development.

The subsequent sections of this paper are organized as follows: Section II introduces the relevant fundamental concepts involved in this study. Section III introduces the *DivSampCA* algorithm, detailing its key components. Section IV describes our experimental settings, while Section V presents the experimental results and provides a comprehensive evaluation of *DivSampCA*'s performance. Section VI briefly reviews the related studies in the field. Finally, Section VII summarizes the paper and presents the conclusions.

II. PRELIMINARIES

A. Boolean Formulae

Given a Boolean variable x, a literal is either x itself or its complement $\neg x$. A clause is composed of literals connected by logical operators such as $\wedge$, $\vee$ and $\neg$. A Boolean formula is usually expressed in *conjunctive normal form* (CNF), where a formula is a conjunction of clauses, and each clause is a disjunction of literals.

An assignment ϕ maps variables to $\{0, 1\}$. For a Boolean formula F, an assignment is complete if every variable in F is assigned a value; otherwise, it is partial. A complete assignment under which F evaluates to true is called a satisfying assignment. If there exists at least one satisfying assignment, then F is satisfiable; otherwise, unsatisfiable.

Example 1. *Consider a CNF formula represented as:*

$$F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4).$$

For the given formula, a satisfying assignment is: $\{x_1, x_2, x_3, x_4\} \rightarrow \{0, 1, 1, 0\}$. *Each clause evaluates to true under this assignment, hence F is satisfiable.*

B. SAT Solving Techniques

MiniSAT is widely utilized in the domain of SAT problem solving [39]. It implements the CDCL (*Conflict-Driven Clause Learning*) framework [40], which iteratively assigns values to unassigned variables and learns new clauses from conflicts to prevent the solver from revisiting identical unsatisfiable states. This process continues until all variables are assigned or a contradictory state is reached.

ContextSAT is a search algorithm that employs SAT solvers to generate batches of valid configurations exhibiting substantial dissimilarity [36]. Built upon the DPLL framework, the algorithm treats the current sample set T as a *context*, computes the frequency with which each variable is assigned 1 in T, and constructs a reference assignment γ through inverse sampling that systematically deviates from the majority values. Subsequently, the algorithm gives precedence to the value assignments specified by γ and randomly permutes the variable order prior to generating each new configuration to preclude re-exploration of any previously traversed search path. The pseudocode for ContextSAT is presented as Algorithm 1.

C. Pairwise Covering Array Generation

A *covering array*, denoted as $CA(N; t, m, v)$, is an $N \times m$ array. The parameter t represents the *strength* of coverage, m denotes the number of options, and v signifies the number of values associated with each option. In all subsequent parts of this paper, v is equal to 2. An $N \times t$ subarray is a matrix that results from choosing t columns out of the m columns, with each row representing a t-wise tuple. The covering array itself ensures that every such $N \times t$ subarray encompasses all possible t-wise tuples of the v values.

Algorithm 1: *ContextSAT* (F, α, γ, π) [36]

Input : F: Boolean formula in CNF;
α: (partial) assignment of F;
γ: a reference assignment of $V(F)$;
π: a variable order of $V(F)$;
Output: solution of F, or reporting "*No Solution*";

if α *is a solution* **then return** α ;
if *No solution can be extended from* α **then return** *"No Solution"* ;
$x \leftarrow$ the first unassigned variable in $V(F)$ according to π;
$D \leftarrow [\gamma[x], 1 - \gamma[x]]$;
foreach σ *in* D **do**
  $\alpha[x] \leftarrow \sigma$;
  $F', \alpha' \leftarrow$ Simplify F and extend α through unit propagation;
  if *ContextSAT*$(F', \alpha', \gamma, \pi)$ *returns a solution* β **then return** β ;
end
return *"No Solution"*

In combinatorial interaction testing, a test suite is typically formatted as a covering array. Each row of the covering array represents a distinct test case. Meanwhile, each column represents an option, and the elements within a column denote the values chosen for that option. When t is set to 2, the array is known as a pairwise covering array, and the PCAG problem refers to the construction of an optimal PCA that meets specific constraints.

III. OUR PROPOSED APPROACH

In this section, we describe the implementation of *DivSampCA*. Inspired by the two-stage greedy framework [34], [36], we integrate tuple-oriented adaptive sampling to generate a diverse set of configurations and cover as many tuples as possible within each configuration for efficient full coverage. Specifically, we first introduce the overall framework of *DivSampCA*, followed by a detailed description of the core algorithmic components.

A. Overall Framework of DivSampCA

The overall framework of the *DivSampCA* algorithm is detailed in Algorithm 2. It takes a CNF formula F, a partial assignment α, and hyperparameters k and *count* as input, and outputs the corresponding PCA if F is satisfiable. Before starting any operation, *DivSampCA* utilizes a preprocessor named *Coprocessor* [41] to simplify the CNF formula. The preprocessor possesses the property of equivalence preservation, which can enhance the solving process's efficiency without compromising correctness. The F in Algorithm 2 actually represents the simplified CNF formula. A valid configuration is essentially a solution to the formula F. In lines 1-3 of the algorithm, *DivSampCA* invokes a *DPLL* procedure to obtain the initial configuration and initializes the configuration sample with it.

Algorithm 2: Overall Framework of *DivSampCA*

Input : F: Boolean formula in CNF;
α : (partial) assignment of F;
k : size of the candidate test case set;
count : threshold for sampling benefit;
Output: T: pairwise covering array of F;

1 **if** DPLL(F, α) *reports "No Solution"* **then return** $\varnothing$;
2 $\alpha \leftarrow$ DPLL(F, α);
3 $T \leftarrow \{\alpha\}$;
4 **while** *True* **do**
  `// Sample a test case set of size k using the TOAS method`
5 $C \leftarrow TOAS(F, k)$;
  // gain(c_i, T) : `number of valid tuples newly covered by the test case` c_i
6 $\beta \leftarrow \arg\max_{c_i} \text{gain}(c_i, T), 1 \le i \le k$;
7 **if** gain$(\beta, T) <$ *count* **then break**;
8 $T \leftarrow T \cup \{\beta\}$;
9 **end**
10 **while** *number of uncovered tuples* > 0 **do**
  `// Construct a test case that covers as many tuples as possible`
11 $\delta \leftarrow MTPC(F)$;
12 $T \leftarrow T \cup \{\delta\}$;
13 **end**
14 **return** *T;*

One can specify the features to be included in the initial configuration by designating the partial assignment α. Subsequently, lines 4-9 of the algorithm represent the sampling phase. *DivSampCA* employs the *TOAS* algorithm to sample k (a hyperparameter) candidate configurations and stores them in $C = \{c_1, c_2, ..., c_k\}$. Following this, it adds the configuration that maximizes the number of newly added valid tuples to the configuration sample. Concretely, the *gain* function in line 6 calculates the number of tuples newly covered by each configuration when it is added to the sample. Then, in line 7, if the number of new tuples added falls short of the hyperparameter *count*, the sampling phase is terminated, and the algorithm transitions into the full coverage phase. Finally, lines 10-14 of the algorithm represent the full coverage phase. *DivSampCA* iteratively invokes the *MTPC* algorithm to construct configurations that cover as many tuples as possible until completing the construction of the PCA. In the actual implementation of the algorithm, different values of the hyperparameters k and *count* influence both the size of the final PCA and the time consumed. Their impact on algorithm performance is experimentally evaluated in Section V-D. The implementations of the *TOAS* and *MTPC* will be detailed in the following two subsections.

B. Tuple-oriented Adaptive Sampling

During the sampling phase, *DivSampCA* adheres to a greedy framework, aiming to identify the configuration that can cover the maximum number of uncovered tuples in each iteration. However, it is unlikely that an efficient algorithm (polynomial-time algorithm) for identifying such cases exists [42]. Therefore, an alternative approach is to generate some configurations that are significantly different from the existing configuration sample, and then select the configuration that can add the maximum number of tuples from them and incorporate it into the sample.

In practical implementation, *DivSampCA* employs a weight function to dynamically update the sampling probabilities of literals. This enables the sampling process to more comprehensively explore the areas that have not been adequately covered, ultimately generating diverse configurations. The intuitive idea behind this is to decrease the sampling probabilities of the literals corresponding to the tuples that have been widely covered. Meanwhile, it increases the probability of sampling the literals corresponding to the uncovered tuples. As a result, the generated configurations are less likely to overlap with the existing configuration sample, thereby achieving the goal of covering more uncovered tuples.

We introduce $W(\cdot)$ as a weight function for a propositional formula F. It maps a variable assignment in F to a non-negative number. Specifically, the weight of variable x being assigned the value σ is calculated as follows:

$$W(x,\sigma) = \sum_{y \in V(F), y \neq x} \frac{\#uncovered(x,y,\sigma)}{\#uncovered(x,y)}, \quad (1)$$

where $V(F)$ is the set of variables in F, and y represents a variable whose potential identities are every variable in F except for x. The notation $\#uncovered(x,y)$ denotes the number of valid uncovered tuples corresponding to variables x and y; $\#uncovered(x,y,\sigma)$ signifies the number of tuples within these valid uncovered tuples in which the variable x is assigned the value σ. In each iteration, *DivSampCA* keeps track of which tuples remain uncovered and updates the weight of each literal to calculate the probability of variable assignments in the next iteration. Precisely, the probability $P(x)$ of variable x being assigned 1 is calculated as follows:

$$P(x) = \frac{W(x,1)}{W(x,1)+W(x,0)}. \quad (2)$$

In the implementation of *DivSampCA*, $\#uncovered(x,y)$ is calculated as 4 (*i.e.*, the possible number of pairwise tuples) minus the number of tuples (x,y) already covered, regardless of whether the remaining uncovered tuples are valid. This is due to the fact that verifying the validity of each tuple before sampling would require a significant amount of time. While a more precise count might intuitively result in superior sampling outcomes, it would also degrade the overall performance of the algorithm.

Algorithm 3: *TOAS* (F,k)

Input : F : Boolean formula in CNF;
k : size of the candidate test case set;
Output: C : a set of test cases of size k;

```
C ← ∅;
for i ← 1 to k do
    foreach x in V(F) do
        P(x) ← 0 ;    // P(x): Probability of x being assigned 1
        foreach σ in [1, 0] do
            W(x, σ) ← 0;
            foreach y in V(F) and y ≠ x do
                u[x, σ] ← number of uncovered tuples for (y, x) with x = σ;
                ω[x] ← number of uncovered tuples for (y, x);
                W(x, σ) ← W(x, σ) + u[x, σ]/ω[x];
            end
        end
        P(x) ← W(x, 1)/(W(x, 1) + W(x, 0))
    end
    γ_i ← sample a assignment according to P(x);
    π_i ← a random variable order;
    β_i ← ContextSAT(F, ∅, γ_i, π_i);
    C ← C ∪ {β_i};
end
return C;
```

Example 2. *Consider a scenario involving four variables x_1, x_2, x_3, and x_4. The uncovered tuples are as follows:*

$$\begin{bmatrix} [x_1=0, x_2=1], \\ [x_1=1, x_2=0], \\ [x_1=0, x_3=0], \\ [x_1=0, x_4=1], \\ [x_1=1, x_4=0], \\ [x_2=1, x_3=1] \end{bmatrix}$$

To determine the probability of each variable being assigned 1 in the next iteration, the weight is calculated as follows:

$W(x_1,1) = \frac{\#uncovered(x_1,x_2,1)}{\#uncovered(x_1,x_2)} + \frac{\#uncovered(x_1,x_3,1)}{\#uncovered(x_1,x_3)} + \frac{\#uncovered(x_1,x_4,1)}{\#uncovered(x_1,x_4)}$
$= 1/2 + 0 + 1/2 = 1$

$\#uncovered(x_1,x_2)$ represents the number of uncovered tuples (x_1,x_2), which equals 2 (namely, $\{x_1=0, x_2=1\}$ and $\{x_1=1, x_2=0\}$). $\#uncovered(x_1,x_2,1)$ represents the number of uncovered tuples (x_1,x_2) where x_1 is assigned the value 1, which equals 1 (i.e., $\{x_1=1, x_2=0\}$). The calculations for other cases are similar.

Thus, the probability of x_1 being assigned 1 is:

$P(x_1) = \frac{W(x_1,1)}{W(x_1,1)+W(x_1,0)} = \frac{1}{1+2} \approx 0.33$

Similarly, the probabilities of x_2, x_3 and x_4 being assigned 1 are:

$P(x_2) = \frac{3/2}{3/2+1/2} = 0.75, P(x_3) = \frac{1}{1+1} = 0.5, P(x_4) = \frac{1/2}{1/2+1/2} = 0.5$

The specific implementation of the *TOAS* algorithm is presented in Algorithm 3. Concretely, we maintain a data structure to track the uncovered tuples in each iteration. As new configurations are selected, this data structure is updated accordingly to reflect the current coverage status. In lines 5-12 of the algorithm, for each variable x, we calculate the weights of x taking the values of 0 and 1 respectively according to Equation 1. Subsequently, in line 13, we determine the probability of assigning each variable the value of 1 in the next iteration in accordance with Equation 2. In line 15, we randomly assign values to each variable based on the sampling probabilities obtained in the previous step, thereby obtaining a reference assignment. However, this reference assignment may not necessarily satisfy the constraints. Consequently, in lines 16-17, we invoke the *ContextSAT* algorithm mentioned in Section II-B to find a valid solution similar to this reference assignment. These steps continues until a set of k configurations is constructed. Through this process, we can construct a number of configurations that are highly distinct from the current sample in each iteration, so as to select the configuration that can add the maximum number of new tuples.

C. More Tuples Per Case

In the full coverage phase, *DivSampCA* needs to ensure that the final sample covers all tuples that do not violate the constraints. Therefore, after the sampling phase, *DivSampCA* continues to add a small number of configurations to achieve full coverage. Notably, *DivSampCA* introduces an innovative technique called *MTPC*. This technique is designed to cover as many tuples as possible with a single configuration, thereby reducing the number of configurations needed to cover the remaining uncovered tuples and minimizing the scale of the final PCA. In the subsequent part of this section, we will discuss the SAT conflict analysis mechanism and the implementation of the *MTPC* method.

Building on the concepts introduced in Section II-A, we delve into the advanced aspects of partial assignment. A partial assignment is a map $\phi : V \to \{0,1\}^{|V|}$, where V is a finite set of variables and $|V|$ represents the cardinality of V. Given a partial assignment, a SAT solver executes the logical reasoning and backtracking search algorithm. The solver will either extend it to a satisfying assignment or claim that the formula is unsatisfiable under the given partial assignment. Leveraging the characteristic that multiple uncovered tuples can coexist simultaneously within the same valid configuration, *DivSampCA* traverses the uncovered tuples. It sequentially adds them to a partial assignment and determines whether this partial assignment can be extended into a complete one. If it can be extended, *DivSampCA* will proceed to add the next uncovered tuple; otherwise, it will skip the current tuple and continue to process the next one. This process will keep going until all uncovered tuples have been processed.

Algorithm 4: *MTPC* (F)

Input: F: Boolean formula in CNF;
Output: δ : a test case for F;

```
1  α ← ∅;
2  U ← the set of uncovered tuples;
3  U ← shuffle(U) ;  // Shuffle the order of the
     uncovered tuples
4  foreach τ in U do
5  |  α ← α ∪ {τ};
6  |  if DPLL(F, α) reports "No Solution" then
7  |  |  α ← α − {τ};
8  |  |  continue;
9  |  end
10 end
11 if DPLL(F, α) returns a solution δ then return δ;
12 return ∅;
```

The detailed pseudocode of *MTPC* is listed in Algorithm 4. Before running the algorithm, *DivSampCA* first collects all valid uncovered tuples. Given that most tuples have already been covered during the sampling phase, this process is less time-consuming. In line 3 of the algorithm, *MTPC* first shuffles the order of the uncovered tuples to reduce potential biases that may arise from a specific sequence. From line 4 to line 10, *MTPC* traverses the uncovered tuples and processes them with the assistance of the *DPLL* program using the method described above. In line 11, the partial assignment is extended to a complete configuration and then integrated into the configuration sample. Subsequently, the information about uncovered tuples is updated to initiate the next round of traversal, and this process continues until all uncovered tuples are fully covered. Throughout this iterative process, *MTPC* constructs configurations that cover multiple uncovered tuples, thereby reducing the size of the final PCA.

Example 3. *Consider a scenario where we have six distinct uncovered tuples, denoted as τ_1 through τ_6. Due to system constraints, the following combinations are prohibited: (τ_1, τ_2), (τ_1, τ_3), (τ_1, τ_5), (τ_1, τ_6), (τ_2, τ_3) and (τ_2, τ_6).*

Applying the MTPC method, we first place τ_1 into a partial assignment. Since τ_1 conflicts with τ_2, τ_3, τ_5, and τ_6, we place τ_4 into the partial assignment and expand it into a complete configuration. Similarly, we identify partial assignments for τ_2 with τ_5, and τ_3 with τ_6. Consequently, we cover them with 3 configurations (assuming they can all be extended to complete assignments).

D. Discussions

1) Main differences between DivSampCA and SamplingCA: SamplingCA updates the variable assignment probabilities by balancing the number of 0s and 1s for each

variable in the configuration sample. Additionally, during the full coverage phase, *SamplingCA* covers only one uncovered tuple each time, while *DivSampCA* identifies the coexisting uncovered tuples so as to maximize the number of tuples covered by a single configuration.

2) Main differences between DivSampCA and CAmpactor: CAmpactor employs a local search framework that iteratively removes redundant configurations while seeking an equivalent PCA of smaller size. Its reliance on a batch-generation paradigm obliges testers to await the completion of the entire configuration set before testing can commence. In contrast, *DivSampCA* supports on-demand, incremental generation: testers can execute the first batch of configurations as soon as it becomes available, thereby reducing test start-up latency.

3) Main differences between DivSampCA and LS-Sampling-Plus: DivSampCA aims to generate a PCA that can cover all valid tuples. In contrast, *LS-Sampling-Plus* focuses on creating configuration sample with high t-wise coverage. Moreover, they have differences in the configuration selection criteria. *DivSampCA* selects configurations that can maximize the number of covered tuples, whereas *LS-Sampling-Plus* chooses configurations based on an approximate scoring function.

IV. EXPERIMENTAL DESIGN

To evaluate our method's performance, we developed a C++ implementation of *DivSampCA*. It is available for public use and accessible online.[1] Before conducting our experiments, we formulated a series of research questions that our experiments are designed to address.

The target of *DivSampCA* is to efficiently generate small-sized PCAs for testing highly configurable systems. This goal leads us to consider two critical aspects of full coverage sampling algorithms: the size of the PCA and the time required for its generation. Consequently, we study the following research questions:

RQ1 How does *DivSampCA* perform compared with its competitors in PCA size and generation efficiency?

We conducted ablation experiments on our two algorithmic components to answer the following questions:

RQ2 Is the *TOAS* method capable of obtaining configurations that cover more tuples?

RQ3 Can the *MTPC* algorithm cover the same tuples with fewer configurations?

Furthermore, we are also interested in the impact of hyperparameters on the performance of our method.

RQ4 What is the impact of hyperparameter k and *count* on the performance of *DivSampCA*?

A. Benchmarks

In our experiments, we utilized a set of 124 publicly available PCAG benchmark instances.[2] These instances originate from a broad, heterogeneous collection of industrial-scale feature models, encompassing operating-system kernels, core utilities [43], and the KConfig ecosystem [30]. The selected PCAG instances exhibit marked heterogeneity in both scale and constraint complexity: the number of variables ranges from 94 to 11,254, while the number of clauses spans 190 to 62,183. The formula density clusters between 2 and 3, situating the instances within the theoretical 3-SAT phase transition region. Furthermore, these public benchmarks have been thoroughly analyzed in extensive research [30], [44], [45], they possess high generality and representativeness.

B. State-of-the-art Competitors

In this study, we conducted experiments by comparing *DivSampCA* with the state-of-the-art algorithms (*i.e.*, *SamplingCA* [36], *CAmpactor* [37] and *LS-Sampling-Plus* [38]) to address the aforementioned questions.

SamplingCA uses sampling techniques to construct a small configuration sample and adds a few valid configurations to achieve full coverage. Extensive experimental results demonstrate that the performance of the *SamplingCA* outperforms all other PCAG algorithms, including *AutoCCAG* [29], *ACTS* [46], and *HHSA* [47]. The source code for *SamplingCA* is publicly accessible.[3]

CAmpactor is an innovative local search algorithm designed to generate small-sized PCAs. Taking a PCA as input, it removes a specific configuration and then searches for another PCA of the resulting size. This process continues until the budget limit is reached. As reported in the literature [37], *CAmpactor* is capable of producing PCAs that are approximately 45% smaller in size compared to those created by current state-of-the-art methods. The implementation of *CAmpactor* is publicly available. [4]

LS-Sampling-Plus iteratively constructs valid configurations based on its novel local search components. According to the experimental results in the literature [38], *LS-Sampling-Plus* can generate configurations with higher coverage than other leading-edge algorithms, such as *NS* [48], *PLEDGE* [49] and *LS-Sampling* [50]. The source code for *LS-Sampling-Plus* can be accessed digitally.[5]

C. Experimental Setup

In this work, all experiments were run on a laptop equipped with an Intel(R) Xeon(R) Gold 6226R CPU, with 503GB of RAM and running Ubuntu 23.10. Considering the probabilistic nature of *DivSampCA* and its competitors, we conducted all experiments 10 times to ensure statistical reliability. Furthermore, we selected *MiniSAT* as the implementation of the *CDCL* algorithm. In the experimental setup for *DivSampCA*, both the hyperparameter k and *count* were set to 100, and for the three competing algorithms, we adopted the hyperparameters recommended in the literature [36]–[38].

[1]https://anonymous.4open.science/r/DivSampCA-D152
[2]https://github.com/edbaranov/feature-model-benchmarks
[3]https://github.com/chuanluocs/SamplingCA
[4]https://github.com/chuanluocs/CAmpactor
[5]https://github.com/chuanluocs/LS-Sampling-Plus

To address RQ1, we evaluated the performance of *DivSampCA* against *SamplingCA* and *CAmpactor* by comparing the average size of the PCAs and the runtime on benchmark instances (with a cutoff time of 3600 CPU seconds). For a strictly fair comparison, all algorithms were uniformly preprocessed with the identical Coprocessor configuration. In addition, following the generation of PCAs by *DivSampCA*, we subsequently employed *LS-Sampling-Plus* to sample configurations of the same size, enabling a comparison of pairwise coverage and generation time.

To address RQ2 and RQ3, we conducted an ablation study to evaluate the contributions of the two core components of *DivSampCA* to the performance improvement. Firstly, *TOAS* strategy was designed to assist in acquiring configurations that cover more tuples during the sampling phase. To assess this, we compared the pairwise tuple coverage achieved by *TOAS* with two other sampling methods (*i.e.*, *Random Sampling* and *SamplingCA*'s Sampling) when generating the same number of configurations. Specifically, across 123 instances excluding `embtoolkit` instance, we generated configurations ranging from 30 to 90 in steps of 10. For the `embtoolkit` instance, the number of configurations ranged from 200 to 900 in increments of 100 due to the requirement of approximately 1000 configurations to achieve full coverage. Secondly, we manipulated the specific sets of remaining uncovered tuples by setting the hyperparameter *count* from 30 to 90 with an increment of 10, and ensured that the content of these tuples remained consistent across all experiments. This allowed us to compare the number of configurations required by *DivSampCA* and *SamplingCA* to cover these identical tuples.

To address RQ4, we varied the values of k to be 10, 20, 100, 200, and 500, and the values of *count* to be 20, 50, 100, 200, and 500 in the final experiment. The purpose of this manipulation was to explore the impact of the interactions between different settings of k and *count* on the performance of *DivSampCA*.

To rigorously assess the performance disparity between *DivSampCA* and the baseline algorithms, we conducted four significance tests on the benchmark instances. When the data exhibited approximate normality, rmANOVA was adopted; otherwise, the Friedman test was applied. Whenever a significant result was obtained, pairwise comparisons were performed using the Wilcoxon signed-rank test with Bonferroni correction, and Cliff's δ was reported as the effect size.

V. EXPERIMENTAL RESULTS

A. Comparisons with State of the Art (RQ1)

Table I and Fig. 1 provide a comprehensive comparison of the performance of *DivSampCA* and its competing algorithms. Given that *CAmpactor* failed to successfully generate PCAs for `embtoolkit` and `uClinux-config` instances, Table I presents the average size and running time of *DivSampCA*, *SamplingCA* and *CAmpactor* algorithms on the entire instance set excluding these two instances. Additionally, it presents the average coverage rate and time of *LS-Sampling-Plus* when generating an equal number of configurations as *DivSampCA*. In Fig. 1, different line styles distinguish the algorithms. Furthermore, the red lines represent the sizes of PCAs and correspond to the left axis, while the blue lines denote the CPU time required for generating PCAs and correspond to the right axis.

TABLE I: The performance of *DivSampCA* (denoted as 'DivSamp'), *SamplingCA* (denoted as 'Samp'), *CAmpactor* (denoted as 'CAmp'), and *LS-Sampling-Plus* (denoted as 'LSP') over the entire set of instances, excluding the `embtoolkit` and `uClinux-config` instances.

	DivSamp	*Samp*	*CAmp*	*LSP*
average size	79.05	105.01	47.07	99.98%
average time (s)	13.51	18.22	226.07	41.66

The experimental results show that in 123 instances out of 124, the PCAs generated by the *DivSampCA* are significantly smaller than those generated by the *SamplingCA*, with an average reduction of approximately 25%. Meanwhile, in 121 out of 124 instances, *DivSampCA* generates PCAs faster than *SamplingCA*, saving an average of 4.71 seconds. In conclusion, the performance of *DivSampCA* is superior to that of *SamplingCA*.

Regarding *CAmpactor*, although *DivSampCA* generates larger PCAs, on average 1.68 times the size of those generated by *CAmpactor*, it only takes 0.06 times the time that *CAmpactor* needs for generation. Moreover, *CAmpactor*'s reliance on batch generation methods rather than incremental ones results in low efficiency in actual testing scenarios. Testers have to wait for a considerable amount of time for the entire configuration sample to be completely generated before starting the testing process, rather than receiving newly generated configurations concurrently during the test execution, which consequently reduces the overall testing efficiency. For *LS-Sampling-Plus*, in 122 out of 124 instances, *DivSampCA* is able to successfully generate configuration sample with 100% coverage and in less time. In contrast, *LS-Sampling-Plus* fails to achieve full coverage and requires a longer time to do so.

Response to RQ1: *DivSampCA* can efficiently generate smaller PCAs compared to the current state-of-the-art PCAG algorithms. Specifically, *DivSampCA* produces PCAs that are roughly 25% smaller and 4.71 seconds faster than *SamplingCA* on average. When compared to *CAmpactor*, although the PCAs generated by *DivSampCA* are 1.68 times larger, the time required is only 0.06 times that of *CAmpactor*. As for *LS-Sampling-Plus*, *DivSampCA* can more efficiently generate configurations with full coverage in 122 out of 124 instances, whereas *LS-Sampling-Plus* takes more time and fails to achieve full coverage.

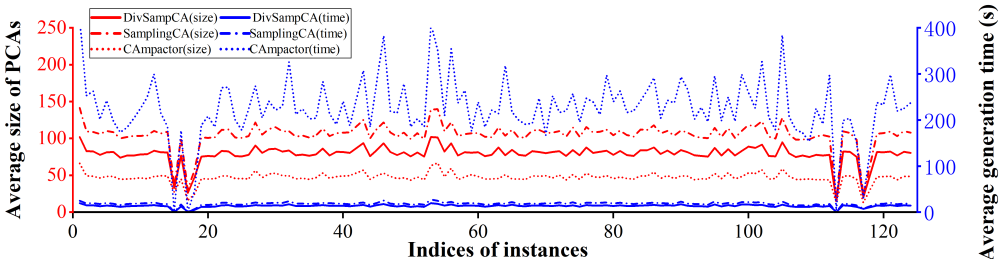


Fig. 1: Performance of *DivSampCA* against *SamplingCA* and *CAmpactor* on the entire instance set.

B. Comparisons of Sampling Strategies (RQ2)

Fig. 2 presents the coverage results of *TOAS* and the other two sampling methods, namely *Random Sampling* and *SamplingCA*'s Sampling. The experimental results show that when generating the same number of configurations, *TOAS* consistently achieves a higher average coverage rate over 123 instances compared with the other two methods. This suggests that the *TOAS* technology can effectively sample more diverse sets of configurations, and obtain configurations that cover more tuples.

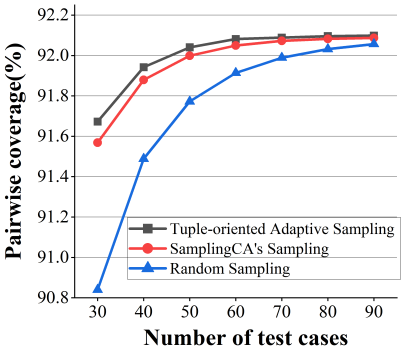


Fig. 2: Average coverage over 123 instances using three sampling methods with an equal number of configurations.

Table II illustrates the proportion of the entire instance set excluding the `embtoolkit` instance in which *TOAS* achieves higher coverage compared to the other two sampling algorithms when generating the same number of configurations. Here, '# tc' represents the number of generated configurations, 'Samp (%)' and 'Rand (%)' denote the proportions in which *TOAS* outperforms *SamplingCA's Sampling* and *Random Sampling*, respectively. The experimental results indicate that in more than 99% of cases, *TOAS* can sample configurations that cover more tuples than randomly generated configurations. Additionally, compared with *SamplingCA*'s Sampling that focuses on individual literals, *TOAS* technique directly optimizes for tuple coverage, thus enabling it to obtain configurations with higher coverage in at least 95% of the instance set.

Response to RQ2: *TOAS* can sample a more diverse set of configurations, thereby enabling the acquisition of configurations that cover more tuples. Compared with the other two methods, *TOAS* consistently achieves a higher average coverage across the entire instance set. Specifically, *TOAS* can generate configurations with higher coverage rates than *Random Sampling* in at least 99% of the instances, and higher coverage rates than *SamplingCA's Sampling* in at least 95% of the instances.

TABLE II: Proportion of instances where *TOAS* achieves higher coverage than two other algorithms over the entire instance set except the `embtoolkit` instance.

# tc	30	40	50	60	70	80	90
Samp(%)	96.77	99.19	100	96.77	95.16	96.77	95.97
Rand(%)	100	99.19	99.19	100	99.19	100	100

C. Effect of the Full Covering Technique (RQ3)

Table III presents the number of configurations required by *DivSampCA* and *SamplingCA's full covering* method (denoted as 'Samp') to cover the same set of tuples using their respective full covering techniques. The results show that the number of configurations required by *MTPC* is significantly less than that of *SamplingCA's full covering* method. As the number of tuples increases from approximately 300 to about 1000, the number of configurations required by *SamplingCA's full covering* method escalates from 3.24 times to 5.52 times that of *MTPC*. This reveals that *MTPC* can indeed identify coexisting tuples among a large number of uncovered tuples, enabling a single configuration to cover more tuples simultaneously, thus reducing the number of configurations required.

TABLE III: Number of configurations required to cover the same set of uncovered tuples using *MTPC* and *SamplingCA's full covering* method, along with their ratio. The number of uncovered tuples is denoted by '# uncov'.

# uncov	321	464	590	721	842	952	1076
Samp	165.74	214.86	253.13	287.89	316.92	343.54	367.63
MTPC	51.13	54.31	56.2	58.95	61.49	64.02	66.56
ratio	3.24	3.96	4.50	4.88	5.15	5.37	5.52

Fig. 3 illustrates the trends in the number of configurations required by two full covering methods as the number of tuples increases. The results indicate that the number of configurations required by both methods

increases with the growth of the number of tuples. However, the growth rate of *MTPC* is significantly slower, merely approximately 1/13 of that of *SamplingCA's full covering* method. This is attributed to the fact that as the number of tuples increases, coexisting tuples become more prevalent. In contrast to *SamplingCA* which directly disregards coexisting tuples, *MTPC* takes them into consideration, enabling it to cover these tuples with fewer configurations. Consequently, appropriately increasing the number of remaining uncovered tuples can enable *MTPC* to play a more pivotal role in reducing the scale of PCAs.

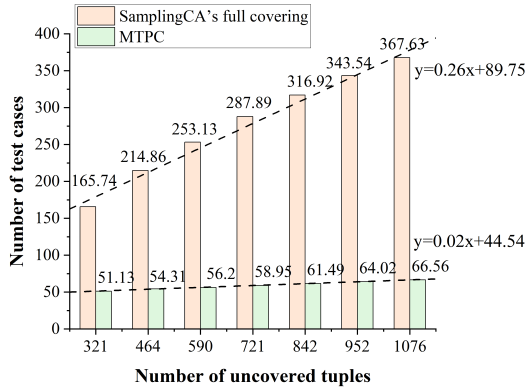
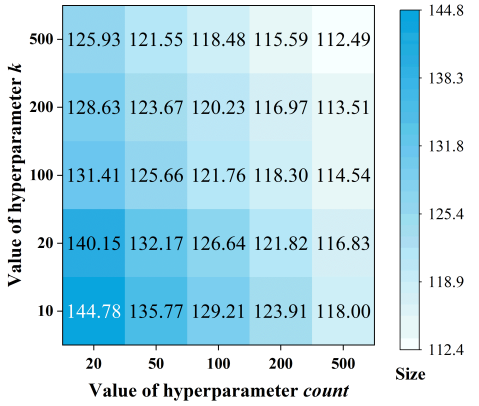


Fig. 3: The number of configurations and their growth rates required by *MTPC* and *SamplingCA's full covering* method to cover the same tuples.

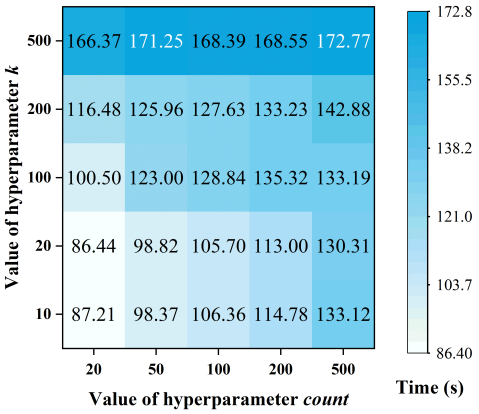
Response to RQ3: Compared to *SamplingCA's full covering* method, *MTPC* requires far fewer configurations to cover the same set of uncovered tuples. This trend becomes more pronounced as the number of uncovered tuples increases. In particular, as the number of tuples increases from approximately 300 to 1000, the number of configurations used by *SamplingCA's full covering* method grows from 3.24 times to 5.52 times that of *MTPC*.

D. Impacts of Hyper-parameter Settings (RQ4)

Fig. 4 presents the average sizes of PCAs and average times of *DivSampCA* under different hyperparameter settings in the form of a heatmap. In Fig. 4a, the average size varies from 112.49 to 144.78, while in Fig. 4b, the average time ranges from 86.44 to 172.77 seconds. The experimental results show that increasing k helps to reduce the size of PCAs, but this typically comes at the cost of increased runtime. Moreover, higher values of *count* can lead to excessive invocations of the SAT solver, thereby increasing time consumption. On the other hand, lower values of *count* may diminish the involvement of *MTPC* during the coverage phase, resulting in larger PCA sizes. In practical testing scenarios, testers can flexibly adjust the values of k and *count* according to specific requirements. The analysis suggests that setting both k and *count* to 100 achieves a trade-off between PCA scale and runtime.



(a)



(b)

Fig. 4: The impacts of hyperparameter on the (a) average size and (b) average time of the *DivSampCA* across the entire instance set. (Darker colors mean worse performance.)

Response to RQ4: Increasing the value of k can effectively enhance the diversity of the configurations, while increasing the value of *count* can boost the participation of *MTPC* technique. These parameter settings reduce the size of PCAs but also increase time consumption. Empirically, setting both k and *count* to 100 achieves a balance that optimizes both efficiency and effectiveness.

E. Statistical Validation Results

The significance tests in Experiment 1 revealed pronounced differences among the algorithms with respect to both configuration size and runtime ($p < 0.0001$). Experiment 2 demonstrates that *ToAS* consistently outperforms both *SamplingCA* (Cliff's δ ranging from 0.043 to 0.01) and the random baseline (δ spanning 0.05–0.54) across all seven sample size levels, while maintaining statistical significance at $p < 0.0001$. In Experiment 3, *MTPC*

significantly surpasses the baseline across all counts with $p < 0.001$ and a median Cliff's δ of 0.92, and its advantage intensifies as scale increases. Finally, Experiment 4 confirms that the main effects of parameters k and *count* on PCA size ($\chi^2 = 20.00, p = 0.0005$) and runtime ($\chi^2 = 18.40, p = 0.0010$) are both statistically significant.

F. Threats to Validity

Internal Validity. To mitigate random error, we conducted multiple independent runs; across 124 benchmark instances, 114 exhibited coefficients of variation below 2% and none exceeded 3.1%. All performance metrics and parameter configurations adopted in this study follow the widely accepted standards and recommended settings documented in the original literature.

External Validity. To enhance the generalizability of the research findings, this study employs a diverse and extensive set of representative benchmark instances. These instances cover a broad range of options and constraints, and have been utilized in numerous previous studies.

VI. RELATED WORK

Combinatorial interaction testing, initially introduced in [52], has been extensively studied in the field of software testing [5]–[7]. In particular, the robust error detection capability of pairwise testing has made it a popular choice for numerous practical applications [6], [14], [53].

The primary objective of pairwise testing is to generate a PCA that covers all pairwise interactions. Existing PCAG algorithms are divided into two categories: exact and approximate algorithms. Exact algorithms aim to construct optimal PCAs [8]–[13], while approximate algorithms focus on generating near-optimal PCAs within an acceptable time. The first exact algorithm, *EXACT* [8], was developed by Yan *et al.* Subsequently, Lopez-Escogido *et al.* proposed another exhaustive search method [13] based on *branch-and-bound* techniques. Building on these earlier efforts, a new algorithm named *IPO-MAXSAT* [10] has been proposed, which encodes the problem into the MaxSAT form [54]. Although exact methods can construct optimal PCAs, their excessively high computational complexity restricts their application in practical scenarios.

Approximate algorithms can generally be categorized into greedy, and meta-heuristic algorithms. Greedy algorithms mainly adopt two strategies: *one-test-at-a-time* (OTAT) [14]–[17] and *in-parameter-order* (IPO) [18]–[22]. Bryce *et al.* proposed a well-known OTAT method named *DDA* [17], which can provide a logarithmic guarantee for the size of the PCA. On the other hand, the IPO algorithm proposed by Lei *et al.* [18] starts from a base case and gradually introduces parameters to ensure that the covering array is expanded both horizontally and vertically. This strategy was further developed in the subsequent *IPOG* [19] and *IPOG-F* [20]. Although greedy algorithms can quickly generate PCAs, they may encounter scalability issues [30], making it difficult to produce small PCAs for large-scale problems.

Meta-heuristic algorithms offer diverse approaches for constructing PCAs, including simulated annealing [23]–[26], genetic algorithms [55], [56] and particle swarm optimization [27], [28], [57]. Torres-Jimenez *et al.* proposed a simulated annealing algorithm dubbed *ISA* [25] and enhanced it in their subsequent work [26]. Thereafter, Garvin *et al.* designed another well-known simulated annealing algorithm called *CASA* [58]. In the field of genetic algorithms, Toshiaki *et al.* combined genetic algorithms with evolutionary strategies to efficiently produce PCAs [55]. Regarding particle swarm optimization approach, Ahmed and Kamal developed a method called *PPSTG* [27]. Based on this strategy, the *PSO-SA* algorithm was proposed to enhance its performance [57]. Meta-heuristic techniques are renowned for their ability to generate small PCAs, but they suffer from issues such as long generation times and inconsistent results.

In this work, we introduce an innovative tuple-oriented adaptive sampling algorithm named *DivSampCA*. To address the inefficiency of existing PCAG algorithms in generating small-scale PCA problems, *DivSampCA* dynamically adjusts sampling probabilities to generate more diverse configurations and achieves full coverage with fewer configurations. Compared to the state-of-the-art algorithms, *DivSampCA* can effectively produce smaller PCAs. This innovative approach marks an advancement in the realm of sampling-based PCAG algorithms.

VII. CONCLUSION

In this work, we propose a tuple-oriented adaptive sampling algorithm called *DivSampCA* to address the efficiency issues of existing PCAG algorithms in generating small PCAs. *DivSampCA* incorporates two effective components, *TOAS* and *MTPC*. By sampling diverse configurations based on the proportion of each literal in the uncovered tuples and utilizing individual configurations to cover more tuples, *DivSampCA* efficiently generates smaller PCAs. The results show that considering the interactions between options during the generation of configurations helps guide sampling to avoid the areas already covered by existing configuration sample, thus facilitating the construction of diverse configurations with less overlap with existing configuration sample. On the other hand, there are co-existence relationships among different tuples. By leveraging the SAT conflict analysis mechanism, the number of configurations required to cover these tuples can be reduced. However, this work is limited by its greedy addition of configurations that cover the maximum number of tuples. Future work will explore the global benefits of covering the tuples more difficult to cover in the early stages. Additionally, a broader range of configuration selection metrics is also an important research direction.

References

[1] A. Von Rhein, A. Grebhahn, S. Apel, N. Siegmund, D. Beyer, and T. Berger, "Presence-condition simplification in highly configurable systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 178–188.

[2] S. Fischer, G. K. Michelon, R. Ramler, L. Linsbauer, and A. Egyed, "Automated test reuse for highly configurable software," *Empirical Software Engineering*, vol. 25, pp. 5295–5332, 2020.

[3] A. Ali, Y. Hafeez, S. Hussain, and S. Yang, "Role of requirement prioritization technique to improve the quality of highly-configurable systems," *IEEE Access*, vol. 8, pp. 27 549–27 573, 2020.

[4] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, Feb. 2011.

[5] I. Dunietzl, C. Mallows, A. Iannino, W. Ehrlich, and B. Szablak, "Applying design of experiments to software testing," in *Proceedings of the (19th) International Conference on Software Engineering*, 1997, pp. 205–215.

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

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

[8] J. Yan and J. Zhang, "Backtracking algorithms and search heuristics to generate test suites for combinatorial testing," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, 2006, pp. 385–394.

[9] J. Bracho-Rios, J. Torres-Jimenez, and E. Rodriguez-Tello, "A new backtracking algorithm for constructing binary covering arrays of variable strength," in *MICAI 2009: Advances in Artificial Intelligence*, A. H. Aguirre, R. M. Borja, and C. A. R. Garciá, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 397–407.

[10] I. Hiess, L. Kampel, M. Wagner, and D. E. Simos, "IPO-MAXSAT: The in-parameter-order strategy combined with MaxSAT solving for covering array generation," in *2022 24th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2022, pp. 71–79.

[11] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2–3, p. 199–219, Jul. 2006.

[12] D. Lopez-Escogido, J. Torres-Jimenez, E. Rodriguez-Tello, and N. Rangel-Valdez, "Strength two covering arrays construction using a SAT representation," in *MICAI 2008: Advances in Artificial Intelligence*, A. Gelbukh and E. F. Morales, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 44–53.

[13] J. Torres-Jimenez, I. Izquierdo-Marquez, A. Gonzalez-Gomez, and H. Avila-George, "A branch & bound algorithm to derive a direct construction for binary covering arrays," in *Advances in Artificial Intelligence and Soft Computing*, G. Sidorov and S. N. Galicia-Haro, Eds. Cham: Springer International Publishing, 2015, pp. 158–177.

[14] D. Cohen, S. Dalal, M. Fredman, and G. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[15] S. Abdullah, Z. Soh, and K. Z. Zamli, "Variable-strength interaction for t-way test generation strategy," *International Journal of Advances in Soft Computing & Its Applications*, vol. 5, no. 3, 2013.

[16] Z. Wang, B. Xu, and C. Nie, "Greedy heuristic algorithms to generate variable strength combinatorial test suite," in *2008 The Eighth International Conference on Quality Software*, 2008, pp. 155–160.

[17] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 159–182, 2007.

[18] Y. Lei and K. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, 1998, pp. 254–261.

[19] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, 2007, pp. 549–556.

[20] M. Forbes, J. Lawrence, Y. Lei, R. Kacker, and D. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrrays," 2008-09-01 00:09:00 2008.

[21] K. Kleine and D. E. Simos, "An efficient design and implementation of the in-parameter-order algorithm," *Mathematics in Computer Science*, vol. 12, pp. 51–67, 2018.

[22] M. Wagner, C. J. Colbourn, and D. E. Simos, "Summary of in-parameter-order strategies for covering perfect hash families," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2023, pp. 268–270.

[23] J. Torres-Jimenez, H. Avila-George, and I. Izquierdo-Marquez, "A two-stage algorithm for combinatorial testing," *Optimization Letters*, vol. 11, no. 3, pp. 457–469, 2017.

[24] J. Torres-Jimenez and E. Rodriguez-Tello, "Simulated annealing for constructing binary covering arrays of variable strength," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[25] ——, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, no. 1, pp. 137–152, 2012.

[26] H. Avila-George, J. Torres-Jimenez, and V. Hernández, "Parallel simulated annealing for the covering arrays construction problem," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 2012, p. 1.

[27] B. S. Ahmed, K. Z. Zamli, and C. Lim, "The development of a particle swarm based optimization strategy for pairwise testing," *Journal of Artificial Intelligence*, vol. 4, no. 2, pp. 156–165, 2011.

[28] T. Mahmoud and B. S. Ahmed, "An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use," *Expert Systems with Applications*, vol. 42, no. 22, pp. 8753–8765, 2015.

[29] C. Luo *et al.*, "AutoCCAG: An automated approach to constrained covering array generation," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 201–212.

[30] T. Pett *et al.*, "Product sampling for product lines: The scalability challenge," in *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, ser. SPLC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 78–83.

[31] J. Lin, S. Cai, C. Luo, Q. Lin, and H. Zhang, "Towards more efficient meta-heuristic algorithms for combinatorial test generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 212–222.

[32] A. Bombarda, A. Gargantini, and A. Calvagna, "Multi-thread combinatorial test generation with smt solvers," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1698–1705. [Online]. Available: https://doi.org/10.1145/3555776.3577703

[33] K. Sarkar, C. J. Colbourn, A. de Bonis, and U. Vaccaro, "Partial covering arrays: Algorithms and asymptotics," in *Combinatorial Algorithms*, V. Mäkinen, S. J. Puglisi, and L. Salmela, Eds. Cham: Springer International Publishing, 2016, pp. 437–448.

[34] C. Luo *et al.*, "Generating pairwise covering arrays for highly configurable software systems," in *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 261–267.

[35] L. Kampel, M. Leithner, B. Garn, and D. E. Simos, "Problems and algorithms for covering arrays via set covers," *Theoretical*

Computer Science, vol. 800, pp. 90–106, 2019, special issue on Refereed papers from the CAI 2017 conference.

[36] C. Luo, Q. Zhao, S. Cai, H. Zhang, and C. Hu, "SamplingCA: effective and efficient sampling-based pairwise testing for highly configurable software systems," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1185–1197.

[37] Q. Zhao *et al.*, "CAmpactor: A novel and effective local search algorithm for optimizing pairwise covering arrays," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 81–93.

[38] C. Luo *et al.*, "Solving the t-wise coverage maximum problem via effective and efficient local search-based sampling," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 1, Dec. 2025.

[39] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518.

[40] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 530–535. [Online]. Available: https://doi.org/10.1145/378239.379017

[41] N. Manthey, "Coprocessor – a standalone SAT preprocessor," in *Applications of Declarative Programming and Knowledge Management*, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 297–304.

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

[43] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 240–251.

[44] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman, "SAT-based analysis of large real-world feature models is easy," in *Proceedings of the 19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 91–100.

[45] C. Luo *et al.*, "Beyond pairwise testing: Advancing 3-wise combinatorial interaction testing for highly configurable systems," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 641–653.

[46] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 540–550.

[47] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 242–251.

[48] Y. Xiang, H. Huang, M. Li, S. Li, and X. Yang, "Looking for novelty in search-based software product line testing," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2317–2338, 2022.

[49] C. Henard *et al.*, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[50] C. Luo *et al.*, "LS-sampling: an effective local search based sampling approach for achieving high t-wise coverage," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1081–1092.

[51] Y. Xiang *et al.*, "Automated test suite generation for software product lines based on quality-diversity optimization," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023.

[52] R. Mandl, "Orthogonal latin squares: an application of experiment design to compiler testing," *Commun. ACM*, vol. 28, no. 10, p. 1054–1058, Oct. 1985.

[53] A. Hervieu, D. Marijan, A. Gotlieb, and B. Baudry, "Practical minimization of pairwise-covering test configurations using constraint programming," *Information and Software Technology*, vol. 71, pp. 129–146, 2016.

[54] C. M. Li and F. Manya, "MaxSAT, hard and soft constraints," in *Handbook of satisfiability*. IOS Press, 2021, pp. 903–927.

[55] T. Shiba, T. Tsuchiya, and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," in *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, 2004, pp. 72–77 vol.1.

[56] S. Sabharwal, P. Bansal, and N. Mittal, "Construction of t-way covering arrays using genetic algorithm," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 2, pp. 264–274, 2017.

[57] Z. Li, Y. Chen, Y. Song, K. Lu, and J. Shen, "Effective covering array generation using an improved particle swarm optimization," *IEEE Transactions on Reliability*, vol. 71, no. 1, pp. 284–294, 2022.

[58] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, pp. 61–102, 2011.