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

Anonymous Author(s)

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 generate a minimal test suite that covers all valid pairwise option value combinations. Existing sampling-based PCAG methods often generate large test suites because they do not adequately focus on directly optimizing pairwise tuple coverage when generating test cases, and they fail to exploit opportunities to use a single test case to cover multiple tuples simultaneously. To address these limitations, we propose *DivSampCA*, which employs a tuple-oriented adaptive sampling technique to enhance the diversity of the generated test suite by directly optimizing tuple coverage. Moreover, it utilizes a full coverage strategy to maximize the number of tuples covered by a single test case, ensuring that all valid pairwise combinations are covered with as few test cases 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.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software product lines*; Search-based software engineering; Software functional properties.

KEYWORDS

Pairwise Testing, Covering Array, Adaptive Sampling

ACM Reference Format:
Anonymous Author(s). 2026. DivSampCA: A Tuple-oriented Adaptive Sampling for Generating Small Pairwise Covering Arrays. In *Proceedings of The 48th International Conference on Software Engineering (ICSE 2026).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ICSE 2026, April 2026, Rio de Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 [20, 69]. A highly configurable system is characterized by numerous configuration options, enabling users to tailor different objects of the program to meet specific needs [3]. Table 1 shows a smart home system with four configuration options. Customers can assign different values to these options to suit their individual needs. For instance, one possible configuration could be `{Control Interface = Touch Screen, Security Setting = Fingerprint, Security Camera Control = Enable, Security Alarm System = Enable}`. This system offers 16 distinct configurations, calculated as 2^4.

In practical scenarios, testing all possible configurations of a configurable system to identify interaction errors between different options is inefficient or even impractical. For a small system as shown in Table 1, 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.

Table 1: A smart home appliance control system.

Control Interface	Security Setting	Security Camera Control	Security Alarm System
Touch Screen	Fingerprint	Enable	Enable
Voice Control	Password	Disable	Disable

To address these challenges, combinatorial interaction testing (CIT) constructs a covering array by selecting values for the configuration options to generate a test suite [54]. A covering array is a collection of test cases that ensures, for a fixed value t, any valid t-wise combinations of option values are included in at least one test case, thereby enabling the detection of errors caused by t-wise 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 [16, 31, 32], highlighting its efficacy as a testing methodology.

In real-world systems, constraints often exist between combinations of options due to complex design requirements [8, 11]. For example, in the system presented in Table 1, the `Security Camera Control` and `Security Alarm System` options must be enabled or disabled simultaneously, implying that test cases containing `{Security Camera Control = Enable, Security Alarm System = Disable}` or `{Security Camera Control = Disable, Security Alarm System = Enable}` are invalid. Table 2 presents a PCA that

Table 2: Pairwise cover array of the system in the Table 1.

Control Interface	Security Setting	Security Camera Control	Security Alarm System
Touch Screen	Fingerprint	Enable	Enable
Touch Screen	Password	Disable	Disable
Voice Control	Fingerprint	Disable	Disable
Voice Control	Password	Enable	Enable

meets these constraints, containing only 4 test cases. Therefore, constructing a PCA that satisfies the system constraints while minimizing its size is of considerable significance for practical testing.

Existing methods for generating PCAs can be broadly categorized into exact (*e.g.*, [7, 25, 26, 41, 65, 74]), greedy (*e.g.*, [1, 9, 10, 21, 30, 33, 34, 70, 71]), meta-heuristic (*e.g.*, [2, 4, 38, 40, 42, 48, 64, 66, 67]), and sampling (*e.g.*, [5, 17, 43, 45–47, 53, 55]) approaches. Exact methods generate optimal PCAs through exhaustive search, but are only feasible for small-scale problems due to their high computational complexity. Greedy algorithms build PCAs by adding one test case at a time or parameter by parameter. Although this approach can generate PCAs quickly, it may struggle with scalability issues [56], making it difficult to generate effective PCAs for large-scale problems. Meta-heuristic techniques, which leverage global search and local optimization strategies, are currently considered the best approach for generating small PCAs, though they can suffer from long processing times [39].

In recent years, advancements in formal reasoning tools have driven extensive research into sampling methods for the efficient construction of PCAs [27, 44, 47, 59]. Existing sampling-based approaches generate diverse test cases by employing different sampling strategies and then select some of them to construct test suites. However, these methods merely focus on diversity and do not directly aim to enhance tuple coverage. They usually set sampling probabilities by only considering the distribution of individual variables while ignoring the interaction information between two-dimensional options. This leads to the situation where the sampled test cases fail to cover more tuples, thus increasing the size of the test suites. On the other hand, we have observed that it is highly inefficient to cover the small number of tuples that are not covered by the current test suite by generating a large number of test cases. This is because these tuples are difficult to be covered through sampling methods, and instead, deliberately designed test cases are required. Additionally, the approach of covering only one tuple with a single test case each time ignores the fact that some tuples can coexist within the same test case, resulting in the need for redundant test cases to achieve full coverage.

To address these limitations, we propose an adaptive sampling algorithm named *DivSampCA*. During the sampling phase, *DivSampCA* introduces a new technique called *tuple-oriented adaptive sampling (ToAS)*. This technique iteratively updates the sampling probabilities based on the proportion of each literal in the uncovered tuples, thereby fostering the generation of diverse test cases that cover a greater number of tuples. During the full coverage phase, *DivSampCA* incorporates an innovative approach dubbed *more tuples per case (MTPC)*, which constructs test cases capable of covering multiple uncovered tuples simultaneously. By leveraging the synergistic interaction between these two core techniques, *DivSampCA* effectively addresses the limitations of existing sampling-based methods, providing a more efficient solution to the 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* [47], *CAmpactor* [75] and *LS-Sampling-Plus* [45]. 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 in less time 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, this paper makes the following contributions:

- We propose a tuple-oriented adaptive sampling technique, termed *ToAS*, which enhances diversity in test suites during the sampling process.
- We develop an efficient full covering technique called *MTPC*, designed to minimize the number of test cases 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 2 presents the foundational concepts related to Boolean formulae, SAT solving techniques and the PCAG problem. Section 3 introduces the *DivSampCA* algorithm, detailing its key components. Section 4 describes our experimental settings, while Section 5 presents the experimental results and provides a comprehensive evaluation of *DivSampCA*'s performance. Section 6 offers a concise overview of related studies in the field. Finally, Section 7 summarizes the paper and presents the conclusions.

2 PRELIMINARIES

2.1 Boolean Formulae

Given a Boolean *variable* x, a *literal* is either a variable x or its complement $\neg x$. A Boolean clause is composed of literals connected by logical operators such as AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). The primary form in which a Boolean formula is structured is the *conjunctive normal form* (CNF), where a formula is a conjunction of clauses, and each clause is a disjunction of literals.

A Boolean variable x can take two values: 0 or 1, and the logical consequences of clauses and formulas can be evaluated based on these values. For a Boolean formula F, an *assignment function* ϕ assigns a value ϕ_i to the variable x_i in F, where ϕ_i is either 0 or 1. A *complete assignment* is an assignment function where every variable in F is assigned a value; otherwise, it is a *partial assignment*. A *satisfying assignment* of F is a complete assignment that makes F evaluate to true. If there exists at least one satisfying assignment of F, then F is satisfiable; otherwise, unsatisfiable.

It is widely recognized that highly configurable software can be modeled as Boolean formulas [6, 53, 60], which can be effectively processed using established techniques [37, 52]. Numerous tools are specifically designed for translating the feature models into CNF formulas (*e.g.*, *Undertaker* [63], *KConfigReader* [19]), and for analyzing them (including *FeatureIDE* [28] and *Z3* [15]). Therefore, the technology for obtaining satisfying assignments for CNF formulas is crucial for testing highly configurable software.

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

$$F = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor x_3 \lor x_4).$$

For the given formula, a satisfying assignment ϕ *is:* $\{x_1, x_2, x_3, x_4\} \rightarrow \{0, 1, 1, 0\}$. *Under this assignment, each clause in the formula is evaluated to be true, confirming that the formula is satisfiable.*

2.2 SAT Solving Techniques

The *Boolean satisfiability* (SAT) problem is a fundamental problem in computer science, where the goal is to determine if there exists an assignment of truth values to a given set of Boolean variables that makes a Boolean formula true.

MiniSAT [18] is a lightweight and efficient SAT solver that has been widely used in the realm of SAT solving research. It employs the *CDCL* mechanism [51], which is an enhancement of the *DPLL* algorithm [13, 14], to perform logical reasoning. It iteratively assigns values to unassigned variables and learns new clauses from conflicts to prevent the solver from revisiting the same unsatisfiable states. This process continues until all variables are assigned or a contradictory state is reached.

ContextSAT [47] is a SAT algorithm designed to generate test cases for configurable systems. Its implementation is based on *MiniSAT* and its pseudocode is shown in Algorithm 1. It takes a CNF formula F, a (partial) assignment α, a reference assignment γ of variables in F (denoted as $V(F)$), and a variable order π as inputs. The output is a solution to formula F that is close to γ. *ContextSAT* specifies a basic assignment and seeks solutions similar to that assignment, which helps increase the diversity of test cases while ensuring their correctness during generation. Its effectiveness has been studied in [47].

2.3 Pairwise Covering Array Generation

A *covering array*, denoted as $CA(N; t, m, v)$, is an $N \times m$ array. Here, 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 [9].

In *combinatorial interaction testing* (CIT), 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* (PCA), and the *pairwise covering array generation* (PCAG) problem refers to the construction of an optimal PCA that meets specific constraints.

Algorithm 1: *ContextSAT* (F, α, γ, π) [47]

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;
$D \leftarrow$ an ordered list based on γ, i.e., $D = [\gamma[x], 1 - \gamma[x]]$;
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"*;

3 OUR PROPOSED APPROACH

In this section, we describe the implementation of *DivSampCA*. Inspired by the two-stage framework [44, 47], we integrate the concept of tuple-oriented adaptive sampling to achieve a diverse set of sampled test cases, and the idea of covering as many tuples as possible within a single test case for efficient full coverage. Specifically, we first introduce the overall framework of *DivSampCA*, followed by a detailed description of the core algorithmic techniques.

3.1 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* [50] 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 test case is essentially a solution to the formula F. In lines 1-3 of the algorithm, *DivSampCA* invokes a *DPLL* algorithm to obtain the initial test case and initializes the test suite with it. One can specify the tuples to be included in the initial test case 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 test cases and stores them in $C = \{c_1, c_2, ..., c_k\}$. Following this, it adds the test case that maximizes the number of newly added valid tuples to the test suite. Specifically, the *gain* function in line 6 calculates the number of tuples newly covered by each test case when it is added to the test suite. 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

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;

// Return a solution of F that is extended from α
1 **if** DPLL(F, α) *reports "No Solution"* **then return** $\emptyset$;
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;*

phase. Finally, lines 10-14 of the algorithm represent the full coverage phase. *DivSampCA* iteratively invokes the *MTPC* algorithm to construct test cases 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 5.4. The implementations of the *ToAS* and *MTPC* will be discussed in detail in the following two subsections.

3.2 Tuple-oriented Adaptive Sampling

During the sampling phase, *DivSampCA* adheres to a greedy framework, aiming to identify the test case 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 [9, 61]. Therefore, an alternative approach is to generate some test cases that are significantly different from the existing test suite, and then select the test case that can add the maximum number of tuples from them and incorporate it into the test suite.

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 test cases. 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 test cases are less likely to

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;

$C \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** k **do**
  // V(F): The set of variables in F
  for x **in** $V(F)$ **do**
    $P(x) \leftarrow 0$; // P(x): Probability of x being assigned 1
    for σ **in** $[1, 0]$ **do**
      $W(x, \sigma) \leftarrow 0$;
      for y **in** $V(F)$ **and** $y \neq x$ **do**
        $u[x, \sigma] \leftarrow$ number of uncovered tuples for (y, x) with $x = \sigma$;
        $\omega[x] \leftarrow$ number of uncovered tuples for (y, x);
        $W(x, \sigma) \leftarrow W(x, \sigma) + u[x, \sigma]/\omega[x]$;
      end
    end
    $P(x) \leftarrow W(x, 1)/(W(x, 1) + W(x, 0))$; // Equation 2
  end
  $\gamma_i \leftarrow$ sample a assignment according to $P(x)$;
  $\pi_i \leftarrow$ a random variable order;
  $\beta_i \leftarrow ContextSAT(F, \emptyset, \gamma_i, \pi_i)$;
  $C \leftarrow C \cup \{\beta_i\}$;
end
return *C;*

overlap with the existing test suite, thereby achieving the goal of covering more uncovered tuples.

We now introduce $W(\cdot)$ as a weight function for a propositional formula F. It maps an assignment of a variable in F to a non-negative number. Concretely, 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.

Example 2. *Consider a scenario where we have 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:
$W(x_1, 1) = \frac{\#uncovered(x_1,x_2,1)}{\#uncovered(x_1,x_2)} + \frac{\#uncovered(x_1,x_3,1)}{\#uncovered(x_1,x_3)} + \frac{\#uncovered(x_1,x_4,1)}{\#uncovered(x_1,x_4)}$
$= 1/2 + 0 + 1/2 = 1$

Here, $\#uncovered(x_1, x_2)$ 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. Specifically, we maintain a data structure to track the uncovered tuples in each iteration. As new test cases 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 2.2 to find a valid solution similar to this reference assignment. Then, in line 18, we add it to the pool of candidate test cases. The entire process continues until a set of k test cases is constructed. Through this process, we can construct a number of test cases that are highly distinct from the current test suite in each iteration, so as to select the test case that can add the maximum number of new tuples.

3.3 More Tuples Per Case

In the full coverage phase, *DivSampCA* needs to ensure that the final test suite covers all tuples that do not violate the constraints. Therefore, after the sampling phase, *DivSampCA* continues to add a small number of test cases 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 test case, thereby reducing the number of test cases needed

Algorithm 4: *MTPC* (F)

Input: F: Boolean formula in CNF;
Output: δ : a test case for F;
1 $\alpha \leftarrow \emptyset$;
2 $U \leftarrow$ the set of uncovered tuples;
3 $U \leftarrow$ shuffle(U) ; // Shuffle the order of the uncovered tuples
4 **for** τ **in** U **do**
5     $\alpha \leftarrow \alpha \cup \{\tau\}$;
6   **if** DPLL(F, α) *reports "No Solution"* **then**
7         $\alpha \leftarrow \alpha - \{\tau\}$;
8   **continue**;
9   **end**
10 **end**
// Extend α into a complete test case δ
11 **if** DPLL(F, α) *returns a solution* δ **then return** δ;
12 **return** $\emptyset$;

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 2.1, we delve into the advanced aspects of partial assignment. A partial assignment is a map $\phi : V \rightarrow \{0, 1\}^{|V|}$, 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 test case, *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.

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 test case and then integrated into the test suite. 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 test cases 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 test case. Similarly, we identify partial assignments for τ_2 with τ_5, and τ_3 with τ_6. Consequently, we cover them with 3 test cases (assuming they can all be extended to complete assignments).

3.4 Discussions

3.4.1 Main similarities and differences between DivSampCA and Baital. Both *DivSampCA* and *Baital* [5] are based on the greedy algorithm framework and adaptively update literal weights to generate diverse test cases. However, *DivSampCA* generates multiple candidate test cases in each iteration and selects the one that maximizes the number of newly covered tuples, while *Baital* generates only one test case per iteration. Most importantly, *DivSampCA* updates the variable assignment probabilities based on the proportion of literals in uncovered tuples, whereas *Baital* updates them according to the proportion of valid tuples corresponding to each literal. Consequently, *Baital* needs to calculate the number of valid tuples corresponding to all literals before sampling, which consumes a great amount of time.

3.4.2 Main similarities and differences between DivSampCA and SamplingCA. Both *DivSampCA* and *SamplingCA* [47] adopt a two-stage algorithm framework and utilize formal reasoning tools, which enables them to efficiently handle highly configurable software with a large number of configuration options. However, *SamplingCA* updates the variable assignment probabilities by balancing the number of 0s and 1s for each variable in the test suite. 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 test case.

3.4.3 Main similarities and differences between DivSampCA and LS-Sampling-Plus. *DivSampCA* and *LS-Sampling-Plus* [45] share the common goal of sampling diverse test cases. However, the problems they aim to solve and their approaches differ. *DivSampCA* is designed to address the PCAG problem, aiming to generate a PCA that can cover all valid tuples. In contrast, *LS-Sampling-Plus* targets the t-wise coverage maximization problem and focuses on creating test suites with high t-wise coverage. Moreover, not only do they have different mechanisms for updating sampling probabilities, but they also have differences in the test case selection criteria. *DivSampCA* selects test cases that can maximize the number of covered tuples, whereas *LS-Sampling-Plus* chooses test cases based on an approximate scoring function.

4 EXPERIMENTAL DESIGN

To analyze the performance of our method, we have developed a C++ implementation of *DivSampCA*. It is available for public use and can be accessed 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 terms of both PCA size and generation efficiency?

Additionally, we assessed our two core technologies and designed ablation experiments to address the following questions:

RQ2 Is the core technology *ToAS* capable of obtaining test cases that cover more tuples?

RQ3 Can the core technology *MTPC* cover the same tuples with fewer test cases?

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*?

4.1 Benchmarks

In our experiments, we utilized a set of 124 publicly available PCAG benchmark instances.[2] These instances are derived from publicly accessible feature models and represent the practical application requirements across various domains and scenarios. The PCAG instances employed in this study exhibited significant variation in both size and constraints. Specifically, the number of variables ranged from 94 to 11,254 and the number of constraint clauses ranged from 190 to 62,183, thereby demonstrating the persuasiveness and universality of our experimental results. Given that these public benchmark instances have been thoroughly analyzed in extensive research [37, 43, 45, 46, 56, 57], they possess high generality and representativeness.

4.2 State-of-the-art Competitors

In this study, we conducted experiments by comparing *DivSampCA* with the state-of-the-art algorithms (*i.e.*, *SamplingCA* [47], *CAmpactor* [75] and *LS-Sampling-Plus* [45]) to address the aforementioned questions.

SamplingCA uses sampling techniques to construct a small test suite and adds a few valid test cases to achieve full coverage. Extensive experimental results demonstrate that the performance of the *SamplingCA* outperforms all other PCAG algorithms, including *AutoCCAG* [42], *FastCA* [38], and *TCA* [40]. 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 test case and then searches for another PCA of the resulting size. This process continues until the budget limit is reached. As reported in the literature [75], *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]

[1] https://anonymous.4open.science/r/DivSampCA-master-0232

[2] https://github.com/edbaranov/feature-model-benchmarks

[3] https://github.com/chuanluocs/SamplingCA

[4] https://github.com/chuanluocs/CAmpactor

LS-Sampling-Plus iteratively constructs valid test cases based on its novel local search components. According to the experimental results in the literature [45], *LS-Sampling-Plus* can generate test suites with higher coverage than other leading-edge algorithms, such as *NS* [72], *PLEDGE* [23] and *LS-Sampling* [46]. The source code for *LS-Sampling-Plus* can be accessed digitally.[5]

4.3 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 *DPLL* 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 [45, 47, 75].

To address RQ1, we evaluated the performance of *DivSampCA* against *SamplingCA* and *CAmpactor* by comparing the average size of the generated PCAs and the runtime on benchmark instances. In addition, after *DivSampCA* had generated the PCAs, we subsequently utilized *LS-Sampling-Plus* to generate a test suite of the same size, enabling a thorough evaluation of their pairwise coverage and generation time.

To address RQ2 and RQ3, we conducted an ablation study to evaluate the contributions of the two core technologies of *DivSampCA* (including *ToAS* and *MTPC* strategies) to the performance improvement. Firstly, *ToAS* strategy was designed to assist in acquiring test cases 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 test cases. Specifically, across 123 instances excluding `embtoolkit` instance, we generated test cases ranging from 30 to 90 in steps of 10. For the `embtoolkit` instance, the number of test cases ranged from 200 to 900 in increments of 100 due to the requirement of approximately 1000 test cases to achieve full coverage. Secondly, we designed experiments to evaluate the effectiveness of the *MTPC* strategy. 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 test cases 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 statistical significance of performance between *DivSampCA* and its competitors, we conducted the Wilcoxon signed-rank test [12]. Additionally, we calculated the Vargha-Delaney effect size [68] to quantify the magnitude of these differences. If all the p-values from the Wilcoxon signed-rank tests at the 95% confidence level are less than 0.05, and the corresponding Vargha-Delaney effect sizes are all greater than 0.7, we consider the performance of *DivSampCA* to be superior to its competitors, and this superiority is deemed statistically significant. In Table 3, the significant results for *DivSampCA* are underlined.

[5]https://github.com/chuanluocs/LS-Sampling-Plus

5 EXPERIMENTAL RESULTS

5.1 Comparisons with State of the Art (RQ1)

Table 3 and Figure 1 provide a comprehensive comparison of the performance of *DivSampCA* and its competing algorithms. Table 3 showcases the average size and running time of *DivSampCA*, *SamplingCA* and *CAmpactor* algorithms on the entire instance set excluding `embtoolkit` and `uClinux-config` instances. Additionally, it presents the average coverage rate and time of *LS-Sampling-Plus* when generating an equal number of test cases as *DivSampCA*. In Figure 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 (in seconds) required for generating PCAs and correspond to the right axis.

Table 3: The performance of *DivSampCA*, *SamplingCA*, *CAmpactor* and *LS-Sampling-Plus* over the entire instance set, excluding the `embtoolkit` and `uClinux-config` instances.

	DivSampCA	*SamplingCA*	*CAmpactor*	*LS-Sampling-Plus*
average size	<u>79.05</u>	105.01	47.07	99.98%
average time (s)	<u>13.51</u>	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 practical testing scenarios, for most feature models, the time required to execute test suites is far longer than that needed to generate them, with each test case taking several hours or even more to execute [73]. The total duration of the testing process is the sum of the time spent on generating and executing test suites. Therefore, although *DivSampCA* only saves a few seconds compared to *SamplingCA* when generating test suites, the size of the generated test suites is reduced by nearly 26 test cases on average. This significantly saves the time for executing test suites and the overall time. 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 test suite to be completely generated before starting the testing process, rather than receiving newly generated test cases 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 test suites with 100% coverage and in less

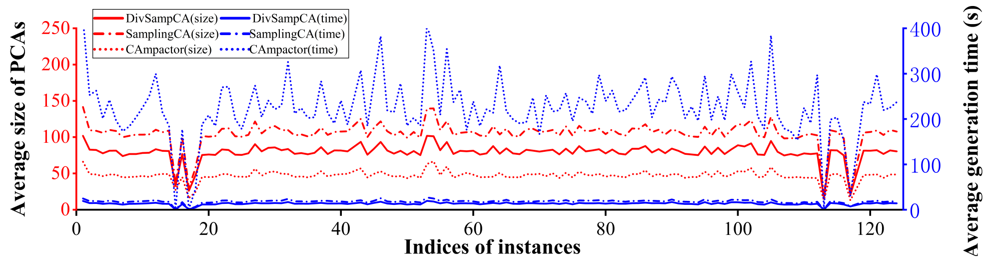

Figure 1: Performance of *DivSampCA* against *SamplingCA* and *CAmpactor* on the entire instance set.

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, compared to *SamplingCA*, *DivSampCA* produces PCAs that are approximately 25% smaller and it is on average 4.71 seconds faster. 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 test suites with full coverage in 122 out of 124 instances, whereas *LS-Sampling-Plus* takes more time and fails to achieve full coverage.

5.2 Comparisons of Sampling Strategies (RQ2)

Figure 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 test cases, *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 test cases, and obtain test cases that cover more tuples.

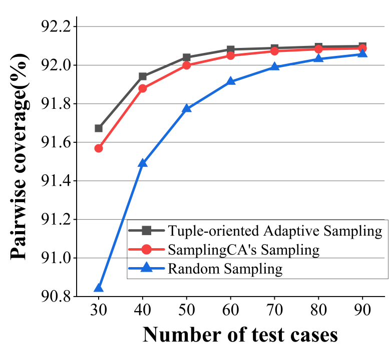

Figure 2: Average coverage over 123 instances using three sampling methods with an equal number of test cases.

Table 4 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 test cases. Here, '# tc' represents the number of generated test cases, 'vs. Samp' and 'vs. 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 test cases that cover more tuples than randomly generated test suites. Additionally, compared with *SamplingCA*'s Sampling that focuses on individual literals, *ToAS* technique directly optimizes for tuple coverage, thus enabling it to obtain test cases with higher coverage in at least 95% of the instance set.

Table 4: 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
vs. Samp(%)	96.77	99.19	100	96.77	95.16	96.77	95.97
vs. Rand(%)	100	99.19	99.19	100	99.19	100	100

Response to RQ2: The core technology *ToAS* can sample a more diverse set of test cases, thereby enabling the acquisition of test cases that cover more tuples. It consistently achieves a higher average coverage across the entire instance set, compared with the other two methods. Specifically, compared to the *Random Sampling*, *ToAS* can generate test suites with higher coverage rates for the same number of test cases in at least 99% of the instances. Compared to *SamplingCA's Sampling*, *ToAS* can generate test suites with higher coverage rates in at least 95% of the instances.

5.3 Effect of the Full Covering Technique (RQ3)

Table 5 presents the number of test cases required by *DivSampCA* and *SamplingCA* to cover the same set of tuples using their respective full covering techniques. The results show that the number of test cases 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 test cases 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 test case to cover more tuples simultaneously, thus reducing the number of test cases required.

Figure 3 illustrates the trends in the number of test cases required by two full covering methods as the number of tuples increases. The results indicate that the number of test cases 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*

Table 5: Number of test cases required to cover the same set of uncovered tuples using *MTPC* and *SamplingCA's full covering* method, along with their ratio. Here, 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

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 test cases. Consequently, appropriately increasing the number of remaining uncovered tuples can enable *MTPC* to play a more pivotal role in reducing the scale of PCAs.

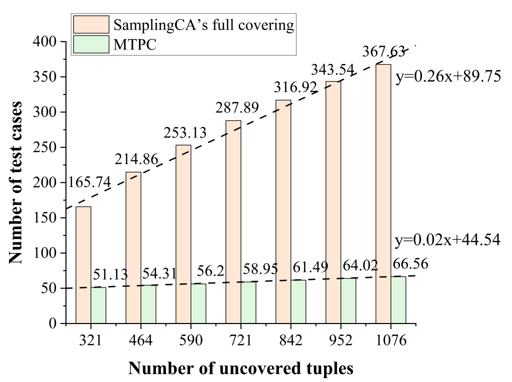

Figure 3: The number of test cases and their growth rates required by *MTPC* and *SamplingCA's full covering* method to cover the same tuples.

Response to RQ3: Compared to *SamplingCA's full covering* method, the number of test cases required by *MTPC* to cover the same set of uncovered tuples is much smaller. 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 test cases used by *SamplingCA's full covering* method grows from 3.24 times to 5.52 times that of *MTPC*.

5.4 Impacts of Hyper-parameter Settings (RQ4)

Figure 4 presents the average sizes of PCAs and average times of *DivSampCA* under different hyperparameter settings in the form of a heatmap. Each row in the figure corresponds to different values of *count* with k remaining unchanged, while each column represents various values of k when the *count* value is constant. In Figure 4a, the average size varies from 112.49 to 144.78, while in Figure 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 efficiency and effectiveness requirements. The analysis suggests that setting both k and *count* to 100 achieves a balance between PCA scale and runtime.

Response to RQ4: According to our experimental results, increasing the value of k can effectively enhance the diversity of the test suite, thereby reducing the size of PCAs, but it also increases time consumption. Furthermore, increasing the value of *count* can increase the participation of $MTPC$ technique, thus reducing the size of PCAs, but it also leads to higher time costs. Empirically, setting both k and *count* to 100 achieves a trade-off that optimizes both efficiency and effectiveness.

5.5 Threats to Validity

Internal Validity. To mitigate the impact of the probabilistic nature of *DivSampCA* on the experimental outcomes, this study conducts multiple experiments. Results show that the coefficient of variation (standard deviation divided by the mean, where a smaller value indicates more stable data) of 114 out of 124 benchmark instances is less than 2%, and the highest is 3.1%. Moreover, the average PCA size in 123 out of 124 benchmark instances is smaller than that of *SamplingCA*, with an average reduction of roughly 25%.
Construct Validity. This study adopts the widely recognized performance metrics from prior research. Specifically, for full coverage algorithms, the size and generation time of PCAs are adopted. For high t-coverage algorithms, t-wise coverage is utilized. Additionally, parameter settings for all methods are based on the recommendations in the literature where they were proposed.
External Validity. To reduce the threat of non-generalizability of the research findings, this study employs a diverse and extensive set of representative benchmarking instances. These instances cover a broad range of options and constraints, and have been utilized in numerous previous studies.

6 RELATED WORK

Combinatorial interaction testing, initially introduced in [49], has been extensively studied in the field of software testing [16, 29, 31, 32]. In particular, the robust error detection capability of pairwise testing (CIT with $t = 2$) has made it a popular choice for numerous practical applications [10, 24, 32].

The core objective of pairwise testing is to generate a PCA that covers all 2-way interactions. Existing PCAG algorithms are divided into two general categories: exact and approximate algorithms. Exact algorithms aim to construct optimal PCAs [7, 25, 26, 41, 65, 74], while approximate algorithms focus on generating near-optimal PCAs within an acceptable time. The first exact algorithm *EXACT* [74] was developed by Yan *et al.* Subsequently, Lopez-Escogido *et al.* proposed another exhaustive search method [65] based on *branch-and-bound* techniques. Recently, a new algorithm named *IPO-MAXSAT* [25] has been introduced, which combines the *in-parameter-order* (IPO) strategy with MaxSAT method [35]. Although exact methods can construct optimal PCAs, their excessively high computational complexity restricts their application in practical scenarios.

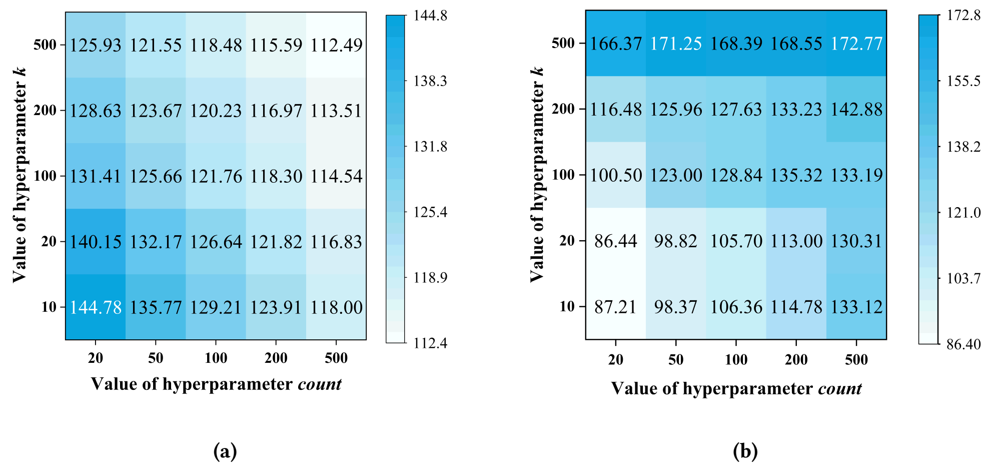

Figure 4: The impacts of hyperparameter settings on (a) average size and (b) average time of the *DivSampCA* across the entire instance set. (Darker colors indicate worse performance.)

Approximate algorithms can be categorized into three types: greedy, meta-heuristic and sampling algorithms. Greedy algorithms mainly adopt two strategies: *one-test-at-a-time* (OTAT) [1, 9, 10, 71] and in-parameter-order [21, 30, 33, 34, 70]. Bryce *et al.* proposed a well-known OTAT method named *DDA* [9], which can provide a logarithmic guarantee for the size of the PCA. On the other hand, the IPO algorithm proposed by Lei *et al.* [34] 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* [33] and *IPOG-F* algorithms [21]. Although greedy algorithms can quickly generate PCAs, they may encounter scalability issues [56], making it difficult to produce small PCAs for large-scale problems.

Meta-heuristic algorithms offer diverse approaches for constructing PCAs, including simulated annealing [4, 64, 66, 67], genetic algorithms [38, 58, 62] and particle swarm optimization [2, 36, 48]. Torres-Jimenez *et al.* proposed a simulated annealing algorithm dubbed *ISA* [67] and enhanced it in their subsequent work [4]. Thereafter, Garvin *et al.* designed another well-known simulated annealing algorithm called *CASA* [22]. In the field of genetic algorithms, Toshiaki *et al.* combined genetic algorithms with evolutionary strategies to efficiently produce PCAs [62]. Regarding particle swarm optimization approach, Ahmed and Kamal developed a strategy called *PPSTG* [2]. Based on this strategy, the *PSO-SA* algorithm was proposed to enhance its performance in generating PCAs [36]. 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.

Sampling methods enable the rapid construction of test cases for systems with numerous configuration options [5, 17, 43, 45–47, 53, 55]. Oh *et al.* studied uniform sampling [55], which samples each valid test case with equal probability. Baranov *et al.* proposed an adaptive weighted sampling method in [5], which covers more valid tuples than uniform sampling. Furthermore, Luo *et al.* proposed two recent cutting-edge PCAG algorithms, namely *SamplingCA* [47] and *LS-Sampling* [46] to mitigate the scalability challenge. An enhanced version of *LS-Sampling*, referred to as *LS-Sampling-plus*, was subsequently proposed in [45]. Although sampling methods can quickly generate PCAs, they either fail to ensure full coverage of all tuples or result in overly large PCAs.

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 test cases and achieves full coverage with fewer test cases. 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.

7 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 test suites based on the proportion of each literal in the uncovered tuples and utilizing individual test cases to cover more tuples, *DivSampCA* efficiently generates smaller PCAs. The results show that considering the interactions between options during the generation of test cases helps guide sampling to avoid the areas already covered by existing test suites, thus facilitating the construction of diverse test cases with less overlap with existing test suites. On the other hand, there are co-existence relationships among different tuples. By leveraging the SAT conflict analysis mechanism, the number of test cases required to cover these tuples can be reduced. However, this work is limited by its greedy addition of test cases 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 test case selection metrics is also an important research direction.

REFERENCES

[1] SA Abdullah, ZH Soh, and Kamal Z Zamli. 2013. Variable-strength interaction for t-way test generation strategy. *International Journal of Advances in Soft Computing & Its Applications* 5, 3 (2013).

[2] Bestoun S Ahmed, Kamal Z Zamli, and C Lim. 2011. The development of a particle swarm based optimization strategy for pairwise testing. *Journal of Artificial Intelligence* 4, 2 (2011), 156–165.

[3] Atif Ali, Yaser Hafeez, Shariq Hussain, and Shunkun Yang. 2020. Role of Requirement Prioritization Technique to Improve the Quality of Highly-Configurable Systems. *IEEE Access* 8 (2020), 27549–27573. https://doi.org/10.1109/ACCESS.2020.2971382

[4] Himer Avila-George, Jose Torres-Jimenez, and Vicente Hernández. 2012. 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 ..., 1.

[5] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: an adaptive weighted sampling approach for improved t-wise coverage. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1114–1126. https://doi.org/10.1145/3368089.3409744

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Software Product Lines*, Henk Obbink and Klaus Pohl (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 7–20.

[7] Josue Bracho-Rios, Jose Torres-Jimenez, and Eduardo Rodriguez-Tello. 2009. A New Backtracking Algorithm for Constructing Binary Covering Arrays of Variable Strength. In *MICAI 2009: Advances in Artificial Intelligence*, Arturo Hernández Aguirre, Raúl Monroy Borja, and Carlos Alberto Reyes Garciá (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 397–407.

[8] Renée C. Bryce and Charles J. Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (2006), 960–970. https://doi.org/10.1016/j.infsof.2006.03.004 Advances in Model-based Testing.

[9] Renée C. Bryce and Charles J. Colbourn. 2007. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability* 17, 3 (2007), 159–182. https://doi.org/10.1002/stvr.365 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.365

[10] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. 1997. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444. https://doi.org/10.1109/32.605761

[11] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. Association for Computing Machinery, New York, NY, USA, 129–139. https://doi.org/10.1145/1273463.1273482

[12] William Jay Conover. 1999. *Practical nonparametric statistics*. Vol. 350. john wiley & sons.

[13] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[14] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215. https://doi.org/10.1145/321033.321034

[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[16] I.S. Dunietzl, C.L. Mallows, A. Iannino, W.K. Ehrlich, and B.D. Szablak. 1997. Applying Design of Experiments to Software Testing. In *Proceedings of the (19th) International Conference on Software Engineering*. 205–215.

[17] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 549–559. https://doi.org/10.1145/3180155.3180248

[18] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, Enrico Giunchiglia and Armando Tacchella (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 502–518.

[19] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the Kconfig semantics and its analysis tools. *SIGPLAN Not.* 51, 3 (Oct. 2015), 45–54. https://doi.org/10.1145/2936314.2814222

[20] Stefan Fischer, Gabriela Karoline Michelon, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2020. Automated test reuse for highly configurable software. *Empirical Software Engineering* 25 (2020), 5295–5332.

[21] Michael Forbes, James Lawrence, Yu Lei, Raghu Kacker, and D. Kuhn. 2008. Refining the In-Parameter-Order Strategy for Constructing Covering Arrrays. https://doi.org/10.6028/jres.113.022

[22] Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16 (2011), 61–102.

[23] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670. https://doi.org/10.1109/TSE.2014.2327020

[24] Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. 2016. Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology* 71 (2016), 129–146. https://doi.org/10.1016/j.infsof.2015.11.007

[25] Irene Hiess, Ludwig Kampel, Michael Wagner, and Dimitris E. Simos. 2022. 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)*. 71–79. https://doi.org/10.1109/SYNASC57785.2022.00021

[26] Brahim Hnich, Steven D. Prestwich, Evgeny Selensky, and Barbara M. Smith. 2006. Constraint Models for the Covering Test Problem. *Constraints* 11, 2–3 (July 2006), 199–219. https://doi.org/10.1007/s10601-006-7094-9

[27] Ludwig Kampel, Manuel Leithner, Bernhard Garn, and Dimitris E. Simos. 2019. Problems and algorithms for covering arrays via set covers. *Theoretical Computer Science* 800 (2019), 90–106. https://doi.org/10.1016/j.tcs.2019.10.018 Special issue on Refereed papers from the CAI 2017 conference.

[28] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*. 611–614. https://doi.org/10.1109/ICSE.2009.5070568

[29] Paris Kitsos, Dimitris E. Simos, Jose Torres-Jimenez, and Artemios G. Voyiatzis. 2015. Exciting FPGA cryptographic Trojans using combinatorial testing. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 69–76. https://doi.org/10.1109/ISSRE.2015.7381800

[30] Kristoffer Kleine and Dimitris E Simos. 2018. An efficient design and implementation of the in-parameter-order algorithm. *Mathematics in Computer Science* 12 (2018), 51–67.

[31] D.R. Kuhn and M.J. Reilly. 2002. An investigation of the applicability of design of experiments to software testing. In *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* 91–95. https://doi.org/10.1109/SEW.2002.1199454

[32] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421. https://doi.org/10.1109/TSE.2004.24

[33] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. 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)*. 549–556. https://doi.org/10.1109/ECBS.2007.47

[34] Yu Lei and K.C. Tai. 1998. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*. 254–261. https://doi.org/10.1109/HASE.1998.731623

[35] Chu Min Li and Felip Manya. 2021. MaxSAT, hard and soft constraints. In *Handbook of satisfiability*. IOS Press, 903–927.

[36] Zhao Li, Yuhang Chen, Yi Song, Kangjie Lu, and Jinwei Shen. 2022. Effective Covering Array Generation Using an Improved Particle Swarm Optimization. *IEEE Transactions on Reliability* 71, 1 (2022), 284–294. https://doi.org/10.1109/TR.2021.3132147

[37] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based analysis of large real-world feature models is easy. In *Proceedings of the 19th International Conference on Software Product Line* (Nashville, Tennessee) *(SPLC '15)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/2791060.2791070

[38] Jinkun Lin, Shaowei Cai, Bing He, Yingjie Fu, Chuan Luo, and Qingwei Lin. 2021. FastCA: An Effective and Efficient Tool for Combinatorial Covering Array Generation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 77–80. https://doi.org/10.1109/ICSE-Companion52605.2021.00040

[39] Jinkun Lin, Shaowei Cai, Chuan Luo, Qingwei Lin, and Hongyu Zhang. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 212–222. https://doi.org/10.1145/3338906.3338914

[40] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. 2015. TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 494–505. https://doi.org/10.1109/ASE.2015.61

[41] Daniel Lopez-Escogido, Jose Torres-Jimenez, Eduardo Rodriguez-Tello, and Nelson Rangel-Valdez. 2008. Strength Two Covering Arrays Construction Using a SAT Representation. In *MICAI 2008: Advances in Artificial Intelligence*, Alexander Gelbukh and Eduardo F. Morales (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–53.

[42] Chuan Luo, Jinkun Lin, Shaowei Cai, Xin Chen, Bing He, Bo Qiao, Pu Zhao, Qingwei Lin, Hongyu Zhang, Wei Wu, Saravanakumar Rajmohan, and Dongmei Zhang. 2021. AutoCCAG: An Automated Approach to Constrained Covering Array Generation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 201–212. https://doi.org/10.1109/ICSE43902.2021.00030

[43] Chuan Luo, Shuangyu Lyu, Qiyuan Zhao, Wei Wu, Hongyu Zhang, and Chunming Hu. 2024. 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* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 641–653. https://doi.org/10.1145/3650212.3680309

[44] Chuan Luo, Jianping Song, Qiyuan Zhao, Yibei Li, Shaowei Cai, and Chunming Hu. 2023. Generating Pairwise Covering Arrays for Highly Configurable Software Systems. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A* (Tokyo, Japan) *(SPLC '23)*. Association for Computing Machinery, New York, NY, USA, 261–267. https://doi.org/10.1145/3579027.3608998

[45] Chuan Luo, Jianping Song, Qiyuan Zhao, Binqi Sun, Junjie Chen, Hongyu Zhang, Jinkun Lin, and Chunming Hu. 2024. Solving the t-wise Coverage Maximum Problem via Effective and Efficient Local Search-based Sampling. *ACM Trans. Softw. Eng. Methodol.* (Aug. 2024). https://doi.org/10.1145/3688836 Just Accepted.

[46] Chuan Luo, Binqi Sun, Bo Qiao, Junjie Chen, Hongyu Zhang, Jinkun Lin, Qingwei Lin, and Dongmei Zhang. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1081–1092. https://doi.org/10.1145/3468264.3468622

[47] Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. 2022. 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* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1185–1197. https://doi.org/10.1145/3540250.3549155

[48] Thair Mahmoud and Bestoun S. Ahmed. 2015. An efficient strategy for covering array construction with fuzzy logic-based adaptive swarm optimization for software testing use. *Expert Systems with Applications* 42, 22 (2015), 8753–8765. https://doi.org/10.1016/j.eswa.2015.07.029

[49] Robert Mandl. 1985. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM* 28, 10 (Oct. 1985), 1054–1058. https://doi.org/10.1145/4372.4375

[50] Norbert Manthey. 2013. Coprocessor – a Standalone SAT Preprocessor. In *Applications of Declarative Programming and Knowledge Management*, Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 297–304.

[51] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2021. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*. ios Press, 133–182.

[52] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference* (San Francisco, California, USA) *(SPLC '09)*. Carnegie Mellon University, USA, 231–240.

[53] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/3336294.3336297

[54] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. https://doi.org/10.1145/1883612.1883618

[55] Jeho Oh, Paul Gazzillo, and Don Batory. 2019. t-wise Coverage by Uniform Sampling. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/3336294.3342359

[56] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 78–83. https://doi.org/10.1145/3336294.3336322

[57] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 240–251. https://doi.org/10.1109/ICST.2019.00032

[58] Sangeeta Sabharwal, Priti Bansal, and Nitish Mittal. 2017. Construction of t-way covering arrays using genetic algorithm. *International Journal of System Assurance Engineering and Management* 8, 2 (2017), 264–274.

[59] Kaushik Sarkar, Charles J. Colbourn, Annalisa de Bonis, and Ugo Vaccaro. 2016. Partial Covering Arrays: Algorithms and Asymptotics. In *Combinatorial Algorithms*, Veli Mäkinen, Simon J. Puglisi, and Leena Salmela (Eds.). Springer International Publishing, Cham, 437–448.

[60] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*. 139–148. https://doi.org/10.1109/RE.2006.23

[61] G. Seroussi and N.H. Bshouty. 1988. Vector sets for exhaustive testing of logic circuits. *IEEE Transactions on Information Theory* 34, 3 (1988), 513–522. https://doi.org/10.1109/18.6031

[62] T. Shiba, T. Tsuchiya, and T. Kikuno. 2004. 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.* 72–77 vol.1. https://doi.org/10.1109/CMPSAC.2004.1342808

[63] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature consistency in compile-time-configurable system software: facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 47–60. https://doi.org/10.1145/1966445.1966451

[64] Jose Torres-Jimenez, Himer Avila-George, and Idelfonso Izquierdo-Marquez. 2017. A two-stage algorithm for combinatorial testing. *Optimization Letters* 11, 3 (2017), 457–469.

[65] Jose Torres-Jimenez, Idelfonso Izquierdo-Marquez, Aldo Gonzalez-Gomez, and Himer Avila-George. 2015. A Branch & Bound Algorithm to Derive a Direct Construction for Binary Covering Arrays. In *Advances in Artificial Intelligence and Soft Computing*, Grigori Sidorov and Sofía N. Galicia-Haro (Eds.). Springer International Publishing, Cham, 158–177.

[66] Jose Torres-Jimenez and Eduardo Rodriguez-Tello. 2010. Simulated Annealing for constructing binary covering arrays of variable strength. In *IEEE Congress on Evolutionary Computation*. 1–8. https://doi.org/10.1109/CEC.2010.5586148

[67] Jose Torres-Jimenez and Eduardo Rodriguez-Tello. 2012. New bounds for binary covering arrays using simulated annealing. *Information Sciences* 185, 1 (2012), 137–152. https://doi.org/10.1016/j.ins.2011.09.020

[68] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[69] Alexander Von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 178–188. https://doi.org/10.1109/ICSE.2015.39

[70] Michael Wagner, Charles J. Colbourn, and Dimitris E. Simos. 2023. Summary of In-Parameter-Order strategies for covering perfect hash families. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 268–270. https://doi.org/10.1109/ICSTW58534.2023.00055

[71] Ziyuan Wang, Baowen Xu, and Changhai Nie. 2008. Greedy Heuristic Algorithms to Generate Variable Strength Combinatorial Test Suite. In *2008 The Eighth International Conference on Quality Software*. 155–160. https://doi.org/10.1109/QSIC.2008.52

[72] Yi Xiang, Han Huang, Miqing Li, Sizhe Li, and Xiaowei Yang. 2022. Looking For Novelty in Search-Based Software Product Line Testing. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2317–2338. https://doi.org/10.1109/TSE.2021.3057853

[73] Yi Xiang, Han Huang, Sizhe Li, Miqing Li, Chuan Luo, and Xiaowei Yang. 2023. Automated Test Suite Generation for Software Product Lines Based on Quality-Diversity Optimization. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 46 (Dec. 2023), 52 pages. https://doi.org/10.1145/3628158

[74] Jun Yan and Jian Zhang. 2006. 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. 385–394. https://doi.org/10.1109/COMPSAC.2006.33

[75] Qiyuan Zhao, Chuan Luo, Shaowei Cai, Wei Wu, Jinkun Lin, Hongyu Zhang, and Chunming Hu. 2023. 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* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3611643.3616284