2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Towards High-strength Combinatorial Interaction Testing for Highly Configurable Software Systems

Chuan Luo
Beihang University
Beijing, China
chuanluo@buaa.edu.cn

Shuangyu Lyu
Beihang University
Beijing, China
shuangyulyu@buaa.edu.cn

Wei Wu
Central South University
Changsha, China
wei.wu@csu.edu.cn

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Dianhui Chu
Harbin Institute of Technology
Weihai, China
chudh@hit.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

***Abstract*—Highly configurable software systems are crucial in practice to satisfy the rising demand for software customization, and combinatorial interaction testing (CIT) is an important methodology for testing such systems. Constrained covering array generation (CCAG), as the core problem in CIT, is to construct a t-wise covering array (CA) of minimum size, where t represents the testing strength. Extensive studies have demonstrated that high-strength CIT (*e.g.,* 4-wise and 5-wise CIT) has stronger fault detection capability than low-strength CIT (*i.e.,* 2-wise and 3-wise CIT), and there exist certain critical faults that can be disclosed through high-strength CIT. Although existing CCAG algorithm has exhibited effectiveness in solving the low-strength CCAG problem, they suffer the severe high-strength challenge when solving 4-wise and 5-wise CCAG, which urgently calls for effective solutions to solving 4-wise and 5-wise CCAG problems. To alleviate the high-strength challenge, we propose a novel and effective local search algorithm dubbed *HSCA*. Particularly, *HSCA* incorporates three new and powerful techniques, *i.e.,* multi-round CA generation mechanism, dynamic priority assigning technique, and variable grouping strategy, to improve its performance. Extensive experiments on 35 real-world and synthetic instances demonstrate that *HSCA* can generate significantly smaller 4-wise and 5-wise CAs than existing state-of-the-art CCAG algorithms. More encouragingly, among all 35 instances, *HSCA* successfully builds 4-wise and 5-wise CAs for 35 and 29 instances, respectively, including 11 and 15 instances where existing CCAG algorithms fail. Our results indicate that *HSCA* can effectively mitigate the high-strength challenge.**

***Index Terms*—Combinatorial interaction testing, local search.**

I. Introduction

The demand for customized software and services is rising, with highly configurable software system playing a crucial role. These software systems provide many configurable options, allowing practitioners to easily customize systems for satisfying various practical requirements [1]–[12]. However, testing such systems is challenging, since the number of possible configurations increases exponentially with the growth in the number of configurable options [4], [7], [8], [12]. Testing all possible configurations would require an impractically long time, creating an urgent need for practical testing methods.

Chunming Hu is the corresponding author of this work.

Combinatorial interaction testing (CIT) is recognized to be an effective method for testing highly configurable systems [1], [4], [7], [8], [12]–[14]. CIT aims to generate a test suite containing a manageable number of configurations (also known as test cases), for uncovering faults caused by combinations of any t options, where t denotes testing strength and is usually a small integer (ranging from 2 to 5) in practice [6]. In the context of CIT, a t-wise tuple is an important notion, referring to a combination of the values of t options. The primary goal of t-wise CIT is to construct a t-wise covering array (CA) that covers all t-wise tuples [1], [6], [15]. Since the size of generated CA (*i.e.,* the number of configurations in the related CA) directly affects the testing efficiency, it is crucial to build a t-wise CA of small size [6], [8], [12]. In practice, real-world configurable systems typically impose constraints, *e.g.,* dependencies and exclusiveness, on interactions among option values [12], [16], [17], and all test cases (*i.e.,* configurations) in a t-wise CA must be valid (*i.e.,* satisfying all constraints), for preventing erroneous test results [8], [12], [17]. The problem of t-wise constrained covering array generation (CCAG) is to build a minimum-sized t-wise CA consisting solely of valid test cases. The CCAG problem is a challenging combinatorial optimization problem [18], [19], effectively solving which still remains a significant challenge [1], [6], [15].

While low-strength CIT (*i.e.,* 2-wise CIT and 3-wise CIT) has exhibited effectiveness in detecting a considerable fraction of faults, there still exist a certain number of critical faults that can only be uncovered through high-strength CIT (*e.g.,* 4-wise CIT and 5-wise CIT) [20]. Also, extensive studies have provided empirical evidence that 4-wise CIT and 5-wise CIT have stronger fault detection capability than 2-wise CIT and 3-wise CIT [20]–[24]. Empirical results in the literature [25] present that 4-wise CIT can detect 89% more faults than low-strength CIT in certain scenarios. An empirical study conducted by Kuhn *et al.* [20] demonstrate that up to about 25% of faults would fail to be detected if high-strength CIT is not applied. Another recent study [24] shows that high-strength CIT can uncover up to 9.54% more

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00113

faults than 3-wise CIT on various real-world configurable systems. Further, in life-critical applications, such as aviation and transportation, neglecting even one single fault could lead to a fatal disaster [6]. For instance, for a widely-adopted traffic collision avoidance system [26]–[28], only 74% of faults can be disclosed by 3-wise CIT, while 4-wise CIT and 5-wise CIT can uncover 89% and 100% of faults, respectively [23]. In addition, 4-wise and 5-wise CIT can detect corner-case faults that cause negative consequences and are challenging to be detected via manual testing [6]. A study conducted by LG Electronics [29] reveals that certain critical faults in washing machines and refrigerators can only be disclosed via 4-wise and 5-wise CIT; more importantly, fixing such faults after the products are sold could cost tens of millions of dollars [29].

There is an urgent need for effective algorithms to solve t-wise CCAG with $t \geq 4$. Practical algorithms for t-wise CCAG can be divided into three major categories, *i.e.,* constraint-encoding algorithms (*e.g.,* [30]–[34]), greedy algorithms (*e.g.,* [8], [18], [35]–[47]), and meta-heuristic algorithms (*e.g.,* [1], [6], [12], [15], [22], [36], [48]–[56]). Existing CCAG algorithms have exhibited effectiveness in building 2-wise and 3-wise CAs. However, when generating t-wise CAs with $t \geq 4$, existing CCAG algorithms suffer from the high-strength challenge [6]; that is, existing CCAG algorithms usually require a considerable amount of time to produce large CAs and may even fail to produce them within a reasonable time.

To the best of our knowledge, there exist very few studies targeting to effectively solve 4-wise and 5-wise CCAG problems. *AutoCCAG* [6] is an effective local search CCAG algorithm, which achieves the state-of-the-art performance when solving 4-wise and 5-wise CCAG. *PRBOT-its* [47] is a recently-proposed method for tackling the high-strength CCAG problem. However, as shown in Section VI, both of them fail to generate 4-wise and 5-wise CAs for many instances within a reasonable time, which urgently calls for effective solutions to 4-wise and 5-wise CCAG problems.

In this work, we propose *HSCA*, a novel and effective local search CCAG algorithm, consisting of generation stage and optimization stage. In the generation stage, *HSCA* proposes a novel multi-round CA generation mechanism, which first builds a 2-wise CA A and then insert new test cases into A to extend A to a 3-wise, 4-wise and eventually a t-wise CA (t denotes the target testing strength). Our empirical study in Section IV-B reveals that a $(t-1)$-wise CA covers the majority of t-wise tuples. Thus, compared to existing CCAG algorithms directly generating high-strength CAs from scratch, *HSCA* handles much fewer tuples when generating the t-wise CA, making *HSCA* better alleviate the high-strength challenge. Besides, in the optimization stage, *HSCA* incorporates two new and powerful techniques, *i.e.,* dynamic priority assigning technique and variable grouping strategy, to enhance its performance.

To evaluate *HSCA*, we conducted extensive experiments on 35 real-world and synthetic instances, which have been widely studied [1], [6], [15], [47], [52], [55], [57]–[59]. Our experiments show that *HSCA* generates significantly smaller 4-wise and 5-wise CAs compared to existing state-of-the-art CCAG algorithms. Notably, out of all 35 instances, *HSCA* successfully builds 4-wise and 5-wise CAs for 35 and 29 instances, respectively, including 11 and 15 instances where existing CCAG algorithms fail.

Our main contributions can be summarized as follows.

- We propose *HSCA*, a novel local search method that alleviates the severe high-strength challenge in CIT.
- *HSCA* incorporates three new techniques, *i.e.,* multi-round CA generation mechanism, dynamic priority assigning technique, and variable grouping strategy, to improve its performance.
- Extensive experiments present that *HSCA* considerably advances the state of the art in solving the high-strength CCAG problem.

II. Preliminaries

A. Combinatorial Interaction Testing

System Under Test. A *system under test* (SUT) S, *i.e.,* a configurable system, can be expressed by a set of *option*s O and *constraint*s H, denoted as $S = (O, H)$. Each option $o_i \in O$ has a value domain R_i. Constraints that specify allowable combinations of option value are captured by H.

Tuple. For an SUT $S = (O, H)$, a *tuple* is defined as a set of options and their values *i.e.,* $\tau = \{(o_{i_1}, r_{i_1}), \cdots, (o_{i_t}, r_{i_t})\}$, indicating that the option $o_{i_j} \in O$ takes value $r_{i_j} \in R_{i_j}$. A tuple of size t is termed as a t-wise tuple.

Test Case. For an SUT $S = (O, H)$, a *test case* σ is a tuple of size $|O|$, *i.e.,* $\sigma \left\{= (o_1, r_1), \cdots, (o_{|O|}, r_{|O|})\right\}$, implying that all options in O are covered, and each $o_i \in O$ takes the value $r_i \in R_i$. A test suite is a collection of test cases. A tuple τ is *covered* by a test case σ if $\tau \subseteq \sigma$, indicating that all options in τ take the same values as in σ; a tuple τ is *covered* by a test suite E if τ is covered by at least one test case in E.

As introduced in Section I, there exist constraints in real-world configurable systems, so the values assigned to options are subject to such constraints [16]. To ensure the correctness of the testing process, each adopted test case must satisfy all constraints. In this work, a test case σ is valid if it satisfies all constraints; a t-wise tuple τ is valid if it is covered by at least one valid test case. Given an SUT S, its test suite E and a testing strength t, the t-wise coverage of E is the ratio between the number of E's covered valid t-wise tuples, and the number of all valid t-wise tuples for S; we note that t-wise coverage is a widely-used metric in CIT [4], [7], [60], [61].

Covering Array. Given an SUT $S = (O, H)$, a t-wise *covering array* (CA) A is a test suite with valid test cases, such that all valid t-wise tuples of S are covered by A. The problem of t-wise CCAG, as a fundamental problem in CIT, is to build a t-wise CA as small-sized as possible [6], [8], [12].

B. Boolean Formula and Model Flattening

1) Boolean Formula: It is recognized that an SUT can be modeled as a Boolean formula [62]–[66]. Recent studies [4], [7], [8], [12] present that an effective approach to deal with highly configurable systems is to employ effective techniques

for handling Boolean formulas. In the following, we introduce Boolean formulas and its related notions.

Given a Boolean *variable* x, either x or $\neg x$ is a *literal*. A *clause* c is a disjunction of literals. Boolean variable is the atom of a Boolean formula, typically expressed in conjunctive normal form (CNF) [67]. A formula F in CNF is a conjunction of clauses, *i.e.,* $F = c_1 \wedge \cdots \wedge c_m$, where c_j $(1 \leq j \leq m)$ is a clause. We use $V(F)$ denoting the set of F's all Boolean variables, and $C(F)$ representing the set of F's all clauses.

For a Boolean variable $x_i \in V(F)$, its value domain is $\{0, 1\}$. An *assignment* ϕ of F is a mapping $\phi : V(F) \rightarrow \{0, 1\}$, where each variable in $V(F)$ is assigned either 0 or 1. Given an assignment ϕ, a clause $c_j \in C(F)$ has two possible *state*s under ϕ: c_j is *satisfied* if there exist any literal in c_j evaluating to be 1 under ϕ; otherwise, c_j is *unsatisfied*. Given an assignment ϕ, if all clauses are satisfied under ϕ, then ϕ is a *satisfying assignment* (also known as a *solution*); otherwise, ϕ is an *unsatisfying assignment*.

2) Model Flattening: Here we briefly introduce the model flattening technique [68], which converts an SUT into a Boolean formula and has been widely applied in recent CIT studies [4], [7], [8], [12]. Given an SUT S, the main idea of model flattening is to use a Boolean variable to represent each possible value of every option within S. For more technical details about model flattening, readers can refer to the literature [68].

Here is an example describing the model flattening process. Given an example SUT S with $O = \{o_1, o_2\}$, the value domains of o_1 and o_2 (*i.e.,* R_1 and R_2) are $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$, respectively. S has one constraint, *i.e.,* $H = \{\neg(o_1 = v_1) \vee \neg(o_2 = v_5)\}$, indicating that o_1 taking value v_1 and o_2 taking value v_5 cannot hold at the same time. After applying model flattening to S, its converted Boolean formula F has 6 Boolean variables (*i.e.,* $V(F) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$) and 9 clauses (*i.e.,*$C(F) = \{\neg x_1 \vee \neg x_5, x_1 \vee x_2 \vee x_3, \neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_2 \vee \neg x_3, x_4 \vee x_5 \vee x_6, \neg x_4 \vee \neg x_5, \neg x_4 \vee \neg x_6, \neg x_5 \vee \neg x_6\}$), where variable x_i represents value v_i. The first clause (*i.e.,* $\neg x_1 \vee \neg x_5$) is to encode the constraint in H, while other 8 clauses are built to ensure that each option can only take one single value in a test case. In this work, such constraints, which are added to represent the "take-only-one-value" requirement, are called *additional constraints*.

Given an SUT S and its converted formula F, a (valid) test case of S is a (satisfying) assignment of F, and a t-wise tuple of S is a combination of F's t literals. For instance, F's one satisfying assignment $\{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 0, x_6 = 1\}$ is the valid test case of S, where option o_1 takes value v_1 and option o_2 takes value v_6; the S's 2-wise tuple, *i.e.,* $\{o_1 = v_2, o_2 = v_4\}$, is the literal combination of F, *i.e.,* $\{x_2, x_4\}$. Given a variable $x_i \in C(F)$, its source option, denoted by $src(x_i)$, is the option of S whose value domain includes value v_i (recalling that x_i corresponds to value v_i). For the above example, we have $src(x_1) = src(x_2) = src(x_3) = o_1$ and $src(x_4) = src(x_5) = src(x_6) = o_2$. Given two different variables, they are neighboring variables of each other when their source options are the same; also, additional clauses only apply to neighboring variables.

Algorithm 1: General Local Search Framework for CCAG

Input: $\boldsymbol{F}$: Boolean formula converted from SUT;
$\boldsymbol{t}$: testing strength;
Output: $\boldsymbol{A^*}$: optimized t-wise CA of F;

```
1 A ← F's t-wise CA, which is built by a greedy algorithm;
2 while termination condition is not met do
3     if A is a t-wise CA then
4         A* ← A;
5         Remove a test case from A;
6         continue;
7     Perform one operation to modify A;
8 return A*;
```

After converting an SUT into a Boolean formula F, the t-wise CCAG problem is equivalent to the problem of constructing a set of F's satisfying assignments, such that all valid t-wise tuples are covered. In theory, finding a satisfying assignment for a Boolean formula is known as the influential Boolean satisfiability (SAT) problem, a typical NP-complete problem [69]. Thus, an efficient SAT solver is required. Recent studies have exhibited the effectiveness of *ContextSAT* [8] in handling Boolean formulas converted from SUTs [8], [12]. Consequently, we adopt *ContextSAT* [8] to handle constraints.

III. Local Search CCAG Algorithms

In this section, we introduce the general framework of local search CCAG algorithms and discuss the limitations.

A. General Framework of Local Search

Meta-heuristic CCAG algorithms can generate smaller-sized CAs than other classes of CCAG algorithms. In practice, state-of-the-art meta-heuristic CCAG algorithms (*e.g.,* [1], [6], [12], [55]) are based on local search [1]. Moreover, local search has recently achieved significant success in solving a variety of other combinatorial optimization problems (*e.g.,* [7], [70]–[81]). Hence, *HSCA* is also designed based on local search.

The general framework of local search CCAG algorithms is outlined in Algorithm 1, consisting of two stages, *i.e.,* generation stage (Line 1 in Algorithm 1) and optimization stage (Lines 2–7 in Algorithm 1). In the generation stage, local search invokes a greedy algorithm (*e.g., SamplingCA* [8] and *ACTS* [44]) to build an initial t-wise CA A. In the optimization stage, local search iteratively performs operations to find a CA of smaller size. Particularly, the main idea of optimization stage is to find a CA of a specific size δ; once a CA of size δ is found, then it attempts to seek a CA of size $\delta - 1$. Thus, during the process, the size of A is decreased.

B. Discussions on Limitations and Solutions

Here we identify the limitations that existing local search CCAG algorithms suffer from. Then we discuss their solutions.

1) High-strength Challenge: As discussed in Section I, existing CCAG algorithms, including local search ones, suffer from the severe high-strength challenge. Specifically, when generating 4-wise and 5-wise CAs, current state-of-the-art CCAG algorithms cannot construct small CAs in an efficient way. The possible reason is that the number of t-wise tuples grows exponentially with the increment in t's value. For example, in a configurable system with 100 options, each having 3 possible values, the number of all possible 3-wise tuples is $\binom{100}{3} \times 3^3$, which is smaller than 5 million. In contrast, the numbers of all possible 4-wise and 5-wise tuples are $\binom{100}{4} \times 3^4$ $\binom{100}{5} \times 3^5$, which exceed 317 million and 18 billion, respectively. Consequently, directly constructing high-strength CAs requires considering an immense number of tuples, resulting in extremely high computational costs.

Solution to High-strength Challenge. *HSCA* proposes a novel *multi-round CA generation mechanism* in the generation stage. In the first round, a 2-wise CA is generated by a greedy algorithm; in subsequent rounds, additional test cases are added to extend it to a 3-wise, 4-wise, and eventually a t-wise CA, where t represents the target testing strength. The intuition behind this solution is straightforward: in each round, only those t-wise tuples that were not covered by the $(t-1)$-wise CA built in the last round need to be considered. According to our empirical analysis in Section IV-B, such uncovered t-wise tuples constitute a very small fraction of all t-wise tuples, significantly reducing computational costs.

2) Tuple Blocking Issue: Given the complex constraints in real-world, highly configurable systems, it is intuitive that a certain number of valid t-wise tuples are difficult to cover, because they can only be covered by specific valid test cases. A recent study [12] reveals that such blocking tuples frequently occur during the search process, making it challenging for existing local search algorithms to find smaller CAs.

Solution to Tuple Blocking Issue. We propose a novel *dynamic priority assigning technique* for enhancing the performance of *HSCA*. Specifically, during the search process, *HSCA* would dynamically assign higher priority to those blocking tuples. Through this way, *HSCA* can be capable of emphasizing blocking tuples, enabling *HSCA* performing those operations that can cover more blocking tuples.

3) Additional Clause Issue: Recent state-of-the-art CCAG algorithms (*e.g.,* [8], [12]) typically handle Boolean formulas converted from given SUTs via model flattening [68]. As described in Section II-B2, apart from the clauses encoded from the constraints of target SUT, model flattening additionally introduces considerably more clauses, for ensuring that each test case satisfies the criterion that every option can only take a single value. Such additional clause issue complicates the process of handling constraints during the optimization stage, which significantly reduces the effectiveness of local search.

Solution to Additional Clause Issue. *HSCA* incorporates a new *variable grouping strategy*, whose main idea is to group all neighboring variables that share the same source option. Using this approach, *HSCA* can identify which variables belong to the same group. As discussed in Section II-B2, additional clauses only apply to variables within the same group, we can design atomic operations that allow *HSCA* to inherently satisfy the "take-only-one-value" requirement. This eliminates the need to handle those additional clauses, thereby improving the performance of local search.

Algorithm 2: Overall Design of our *HSCA* Algorithm

Input: $\boldsymbol{F}$: Boolean formula converted from SUT;
$\boldsymbol{t}$: testing strength;
Output: $\boldsymbol{A^*}$: optimized t-wise CA of F;

```
1  A ← a 2-wise CA of F, which is generated by SamplingCA;
2  A' ← A;
3  for t' ← 3 to t do
4      U ← set of valid t'-wise tuples not covered by A';
5      E ← Extension(t', U);
6      A' ← A' ∪ E;
7  E ← Optimization(F, E, U);
8  A' ← A' ∪ E;
9  U ← the set of all valid t-wise tuples;
10 A* ← Optimization(F, A', U);
11 return A*;
```

IV. Our Proposed *HSCA* Algorithm

In this section, we propose and describe *HSCA*, a novel and effective algorithm for generating high-strength CAs.

A. Overall Design of HSCA

Following the general framework of local search CCAG algorithms (as described in Section III-A), the overall design of *HSCA* is presented in Algorithm 2. There are two inputs for *HSCA*: a) F, the Boolean formula converted from a given SUT, and b) t, the target testing strength. The output of *HSCA* is an optimized t-wise CA of F, denoted by A^*.

According to Algorithm 2, *HSCA* consists of two stages, *i.e.,* generation stage and optimization stage. In the generation stage (Lines 1–6 in Algorithm 2), *HSCA* generates a t-wise CA A in a multi-round manner. In the optimization stage (Lines 7–10 in Algorithm 2), *HSCA* conducts local search to reduce the size of t-wise CA A, which is built in the generation stage, resulting in an optimized t-wise CA A^*.

We note that *HSCA* is an anytime algorithm, which gradually reduces the size of the t-wise CA as the searach process progresses. Once terminated, the smallest-sized t-wise CA found so far is returned as the final output.

B. Generation Stage

In this subsection, we introduce *HSCA*'s generation stage.

1) Motivation and Empirical Study: As introduced in Section III-B, existing CCAG algorithms suffer from the serious high-strength challenge, urgently calling for practical solution.

Key Observation. A recent study [7] shows that a test suite with high t-wise coverage achieves high $(t+1)$-wise coverage. Therefore, it is reasonable to infer that a $(t-1)$-wise CA (*i.e.,* a test suite with 100% $(t-1)$-wise coverage) covers the majority of valid t-wise tuples, and we conducted an empirical study to validate this speculation. Particularly, we adopted 5 well-studied, real-world instances (*i.e.,* configurable systems) as our

TABLE I
t-WISE COVERAGE ACHIEVED BY $(t-1)$-WISE COVERING ARRAYS.

	$t=3$	$t=4$	$t=5$
t-wise coverage achieved by *AutoCCAG*'s $(t-1)$-wise covering array	91%	99%	98%
t-wise coverage achieved by *CAmpactor*'s $(t-1)$-wise covering array	93%	99%	99%

subjects, and these instances have been broadly evaluated in the literature [1], [6], [15], [52], [55]; then, we use *AutoCCAG* [6] and *CAmpactor* [12], two of the current state-of-the-art CCAG approaches, as the CA generation methods in this empirical study. Both methods are described in Section V-B. In this empirical study, each algorithm run for generating t-wise CAs ($2 \leq t \leq 4$) is limited to a cutoff time of 1,000 seconds; instances where t-wise CAs cannot be generated within this time limit are excluded from the computation of the average t-wise coverage. The related empirical results are presented in Table I. The average t-wise coverage obtained by both *AutoCCAG* and *CAmpactor* across all testing strengths is greater than 90%. Our empirical results indicate that $(t-1)$-wise CA can effectively cover the majority of valid t-wise tuples in real-world scenarios. This key observation motivates us to build t-wise CAs based on $(t-1)$-wise CAs, using a novel multi-round mechanism to progressively increase the testing strength.

2) Multi-round CA Generation Mechanism: Given the target testing strength t, the main idea of our multi-round CA generation mechanism is to first build a 2-wise CA and then extend it to a 3-wise, 4-wise and ultimately a t-wise CA. The effectiveness of our multi-round mechanism is based on the above observation that a $(t-1)$-wise CA covers the majority of t-wise tuples. Compared to directly constructing a t-wise CA, our mechanism (*i.e.,* extending from low-strength CA) only needs to handle those t-wise tuples that remain uncovered, thereby considerably reducing the problem complexity.

For building the initial 2-wise CA A, *HSCA* invokes an efficient greedy algorithm *SamplingCA* [8], since recent studies [8], [56], [82] have shown its effectiveness for building 2-wise CAs. Then, *HSCA* gradually extends A into a t-wise CA.

In each round, notation A' denotes the $(t'-1)$-wise CA generated in the last round, where t' represents the testing strength to be achieved in the current round. The primary target of the current round is to extend A' into a t'-wise CA through adding new test cases. Particularly, in each round, *HSCA* first obtains the set of all valid t'-wise tuples that are not covered by A', denoted by U'; then, *HSCA* proposes an extension method (which will be introduced in Section IV-B3) to produce a test suite E that covers all tuples in U. In this way, it is clear that $A' \cup E$ is a t'-wise CA, since it is guaranteed to cover all t'-wise tuples. After the current round is finished, *HSCA* starts the next round, whose target is to further extend $A' \cup E$ into a $(t'+1)$-wise CA. To this end, when constructing the t'-wise CA, compared to directly considering all valid t'-wise tuples, our multi-round mechanism only needs to handle those t'-wise tuples in U, which represent a small proportion of all valid t'-wise tuples (as discussed in Section IV-B1).

Algorithm 3: *Extension* Method

Input: $\boldsymbol{t'}$: testing strength to be obtained in current round;
$\boldsymbol{U}$: set of all uncovered, valid t'-wise tuples;
Output: $\boldsymbol{E}$: test suite that covers all t'-wise tuples in U;

```
1  E ← ∅;
2  U' ← U;
3  while True do
4      Γ ← set of valid test cases by constraint-aware sampling;
5      σ* ← the test case with the largest benefit in Γ;
6      if benefit(σ*, U') ≤ 0 then break;
7      E ← E ∪ {σ*};
8      Remove t'-wise tuples covered by σ* from U';
9  foreach t'-wise tuple τ in U' do
10     Generate a valid test case σ that covers τ;
11     E ← E ∪ {σ};
12     Remove t'-wise tuples covered by σ from U';
13 return E;
```

Once the t-wise CA (t denotes the target testing strength) is obtained, our multi-round mechanism terminates. Hence, in contrast to existing algorithms that directly build t-wise CA, this mechanism significantly reduces the number of tuples to consider in each round, enhancing the performance of *HSCA*.

3) Extension Method: Our extension method, outlined in Algorithm 3, takes t' and U as inputs and outputs a set of valid test cases covering all t'-wise tuples in U, denoted by E. We note that E is initialized as an empty set, and notation U' stands for the current set of t'-wise tuples that belong to U and are not covered by E. Our extension method is comprised of two procedures: sampling procedure and insertion procedure. In the sampling procedure (Lines 1–8 in Algorithm 3), *HSCA* employs an effective constraint-aware sampling technique [7], [8] to build a test suite that contains valid test cases and covers the majority of t'-wise tuples in U. In the insertion procedure (Lines 9–12 in Algorithm 3), *HSCA* inserts a few test cases into E, ensuring that E covers all t'-wise tuples in U.

In the sampling procedure, *HSCA* performs an iterative process to build the test suite. In each iteration, a valid test case σ is generated and added into E. Since E's cardinality directly impacts the size of final CA, it is essential to add those test cases, which cover a large number of tuples in U', into E, thereby reducing E's cardinality. Hence, we design an evaluation metric named $benefit$ to quantify the actual benefit of a given test case if it is added into E. Given a test case σ, its *benefit*, denoted by $benefit(\sigma, U')$, is computed as the number of t'-wise tuples that are covered by σ and belong to U'. Inspired by recent studies [7], [8], *HSCA* generates test cases with large benefit in a greedy manner. In each iteration, a candidate set Γ of valid test cases is built through an effective constraint-aware sampling technique [8]. According to the literature [8], the test cases constructed by the constraint-aware sampling technique are guaranteed to be valid and of large benefits. Then, from Γ *HSCA* selects the test case σ^* with the largest $benefit$ and adds σ^* into E. As the iterative

Algorithm 4: *Optimization* Approach

Input: $\boldsymbol{E}$: test suite to be compacted;
$\boldsymbol{U}$: set of t-wise tuples that need to be covered;
$\boldsymbol{F}$: Boolean formula converted from SUT;
Output: $\boldsymbol{E^*}$: optimized test suite covering all tuples in U;

```
1  E* ← E;
2  foreach tuple τ in U do
3      priority(τ) ← 1 ;
4  while termination condition is not met do
5      if E covers all tuples in U then
6          E* ← E;
7          Remove a test case from E;
8          continue;
9      I ← the candidate set of feasible atomic operations;
10     if I ≠ ∅ then
11         op* ← the operation with the largest score from I;
12         if score(op*) > 0 then
13             Perform operation op* on E;
14             continue;
15     foreach uncovered tuple τ in U do
16         priority(τ) ← priority(τ) + 1;
17     τ ← randomly select an uncovered tuple from U;
18     σ ← randomly select a test case from E;
19     σ' ← invoke ContextSAT to build a test case covering τ;
20     Perform operation op = (σ, σ') on E;
21 return E*;
```

process continues, the number of uncovered tuples in U' decreases, reducing the benefits of selected test cases. The iterative process terminates once the benefit of the selected test case becomes 0.

Although the test suite E built in the sampling procedure covers the majority of t'-wise tuples in U, there still remain few uncovered tuples (*i.e.,* those tuples in U'). Thus, in the insertion procedure, for each uncovered tuple τ in U', *HSCA* invokes a powerful constraint solver called *ContextSAT* [8] (as described in Section II-B2) to generate a valid test case σ that covers τ and then inserts σ into E. Since it is possible that σ would cover more than one tuple in U', *HSCA* would remove such tuples, which are covered by σ and belong to U', from U'. Through this way, our extension method is able to build a test suite E that covers all t'-wise tuples in U.

C. Optimization Stage

Once the generation stage terminates, a $(t-1)$-wise CA, denoted by A', a set of t-wise tuples not covered by A', denoted by U, and a test suite that covers all t-wise tuples in U, denoted by E, are obtained. As discussed in Section IV-B2, $A' \cup E$ is actually a t-wise CA. During the optimization stage, the primary target of *HSCA* is to reduce the size of the t-wise CA that is generated in the generation stage.

According to *HSCA*'s overall design (presented in Algorithm 2), in the optimization stage *HSCA* performs two passes of optimization approach, which will be introduced in Section IV-C. Particularly, the first pass of optimization focuses on compacting E, ensuring that the tuples in U' remain covered by E (Lines 7 in Algorithm 2). In this pass, *HSCA* only needs to deal with the tuples in U', which constitute only a small proportion of all valid t-wise tuples (as discussed in Section IV-B1); Thus, the problem complexity is considerably reduced, so E can be compacted effectively. Since the first pass does not compress A', there might be redundancy within A'. As a result, the second pass is designed to decrease the size of entire CA (*i.e.,* $A' \cup E$), while keeping that all t-wise tuples are covered (Line 10 in Algorithm 2). Through the second pass, the size of the final t-wise CA can be further reduced.

As outlined in Algorithm 4, our optimization approach has two inputs: a) E, the test suite to be compacted, and b) U, the set of t-wise tuples that need to be covered. The primary target of optimization approach is to reduce the size of E, while guaranteeing that all tuples in U remain covered by E. According to the general framework of local search CCAG algorithms (as outlined in Algorithm 1), there are two key components in local search: a) the termination condition, and b) the operation determination mechanism. We first discuss the termination conditions of *HSCA*'s two passes of optimization. In the first pass of optimization, the search process terminates if *HSCA* cannot find a smaller t-wise CA within L consecutive search steps. Here L is a hyper-parameter of *HSCA*, whose effect will be empirically analyzed in Section VI-C. In the second pass of optimization, the search process stops if the running time of *HSCA* exceeds the allowed time budget.

Here we introduce the operation determination mechanism, which is critical in *HSCA*. Inspired by the two-mode local search paradigm that has exhibited effectiveness in solving various combinatorial optimization problems (*e.g.,* [72], [73], [83]–[85]), *HSCA* also adopts this two-mode operation determination mechanism, and in each search step *HSCA* works between greedy mode (Lines 5–16 in Algorithm 4) and diversification mode (Lines 17–20 in Algorithm 4). In the greedy mode, *HSCA* targets to perform such operations that can optimize the objective (*i.e.,* reducing the size of E while covering all tuples in U). In the diversification mode, *HSCA* aims to conduct those operations that can diversify the search direction, for exploring the promising search space. Through adopting this two mode operation determination mechanism, *HSCA* can achieve a good balance between greediness and diversification, thereby enhancing its practical performance. The technical details of both modes are introduced below.

D. Greedy Mode

In this work, given the current test suite E to be compacted, an operation is defined as a pair of two test cases, *i.e.,* $op = (\sigma, \sigma')$, where $\sigma \in E$ and $\sigma' \notin E$, meaning that test case σ is replaced with σ' in E. An operation $op = (\sigma, \sigma')$ is *feasible* if the newly-added test case σ' is valid. As discussed in Section IV-C, the primary target of greedy mode is to perform those operations that can explicitly optimize the objective. According to state-of-the-art local search CCAG algorithms [6], [12], a typical greedy mode works as follows: first, a candidate set I of feasible operations is constructed; then, from I the best operation (*i.e.,* the operation that can make the largest contribution to optimize the objective) is selected

and performed. Thus, in the greedy mode there are two critical problems: a) how to effectively quantify the contribution of an operation op; b) how to construct a candidate set of feasible operations. The solutions to both problems are presented and discussed in Sections IV-D1 and IV-D2, respectively.

1) Dynamic Priority Assigning Technique: To quantify the contribution of a given operation, current state-of-the-art local search CCAG algorithms (*e.g., AutoCCAG* [6] and *CAmpactor* [12]) adopts a straightforward scoring function [1], which directly assesses an operation op via calculating the increment in the number of covered t-wise tuples in U if op is performed. As discussed in Section III-B2 and in the literature [12], the presence of complex constraints leads to a certain number of blocking tuples that are difficult to cover and frequently appear during the search process. However, the straightforward scoring function described above does not distinguish between different tuples, causing current state-of-the-art local search CCAG algorithms to be unaware of these blocking tuples. As a result, these algorithms tend to perform operations that cover more tuples overall but fail to cover the blocking tuples. Thus, due to the existence of blocking tuples, current state-of-the-art local search CCAG algorithms would be trapped into local optima, causing its search process to stagnate.

To address this problem, we propose a novel dynamic priority assigning technique, which aims to emphasize the blocking tuples. Specifically, each t-wise tuple τ in U is assigned an integer as its priority value, denoted by $priority(\tau)$. Then, we use priority value to measure the difficulty of covering a tuple. That is, given two different t-wise tuples τ_1 and τ_2, if τ_1 has a greater priority value than τ_2, then τ_1 is considered more difficult to cover than τ_2. To this end, those blocking tuples would be assigned large priority values.

When to Update Priority Values. Here we discuss when to update priority values for tuples in U. The priority values for all tuples in U are initialized as 1 (Lines 2–3 in Algorithm 4), and would be dynamically updated. Since the main idea is to use priority values to differentiate the difficulty of covering different tuples, it is advisable to update priority values when *HSCA* encounters local optima (*i.e.,* there is no operation that can positively contribute to optimizing the objective). Particularly, when *HSCA* gets stuck in local optima, it is intuitive that the tuples that still remain uncovered can be considered to difficult to cover, so the priority value of each uncovered tuple in U is increased by 1 (Lines 15–16 in Algorithm 4). Through this way, the priority values can effectively reflect the difficulty of covering different tuples.

How to Use Priority Values. In order to make *HSCA* be able to perform operations that can cover blocking tuples, it is advisable to design a new priority-based scoring function, which evaluates an operation's contribution based on priority values. Particularly, given an operation op, its *score*, denoted by $score(op)$, is computed as the increment in the total priority values of covered t-wise tuples in U if op is performed. Since blocking tuples have greater priority values than other tuples, through incorporating the priority-based scoring function, *HSCA* tends to perform such operations that can cover more blocking tuples. Therefore, our dynamic priority assigning technique can address the tuple blocking issue (as described in Section III-B2) in existing local search CCAG algorithms.

2) Variable Grouping Strategy: As discussed in Sections II-B2 and III-B3, the widely-used model flattening technique [68], which converts an SUT into a Boolean formula, introduces significantly more clauses to guarantee that each test case in E satisfies "take-only-one-value" requirement. Clearly, these additional clauses complicate the process of handling constraints, which makes the generation of valid test cases more challenging. To build valid test cases, existing CCAG algorithms employ a constraint solver to handle those constraints [8], [12]. However, it is known that invoking a constraint solver one time costs a certain amount of running time [8]. Consequently, for existing CCAG algorithms, the constraint solver-based method [8], [12], which is employed by existing CCAG algorithms, is time-consuming, causing performance degradation when solving the CCAG problem.

To mitigate this issue, we propose a new variable grouping strategy. As discussed in Section II-B2, additional clauses apply only to neighboring variables (*i.e.,* variables sharing the same source option). Our variable grouping strategy clusters all neighboring variables in the same group, enabling the identification of which variables belong to the same group. We use notation G to denote the collection of all groups, and the number of groups (*i.e.,* $|G|$) is equal to the number of options in the target SUT. Using our variable grouping strategy, we can design atomic operations that ensure all additional clauses are satisfied without the need to call a constraint solver.

Atomic Operation. In this work, an atomic operation $op = (\sigma, \sigma')$ is an operation where the test case σ' is derived from the test case σ by simultaneously changing the values of two variables within the same group g, while ensuring that σ' satisfies $\sum_{x_i \in g} x_i = 1$ (*i.e.,* for σ', exactly one variable in group g takes the value of 1). Given a test case σ and a group g that contains p variables, we can easily derive $p-1$ atomic operations with respect to σ and g. As an illustrative example, assuming $\sigma = \{x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0, x_5 = 1\}$ and $g = \{x_1, x_2, x_3\}$, we can obtain 2 atomic operations with respect to σ and g, *i.e.,* $op_1 = (\sigma, \sigma_1)$ and $op_2 = (\sigma, \sigma_2)$, where $\sigma_1 = \{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0, x_5 = 1\}$ and $\sigma_2 = \{x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1\}$.

Building a Candidate Set of Feasible Operations. Given a test case σ and a group g, we use notation $Q(\sigma, g)$ to denote the collection of feasible atomic operations with respect to σ and g. Actually, in the greedy mode we can easily build a candidate set of feasible atomic operations, *i.e.,* $I = \bigcup_{\sigma_i \in E, g_j \in G} Q(\sigma, g)$, which is the union of all possible collections of feasible atomic operations. In the greedy mode, once the candidate set of feasible atomic operations I is built, *HSCA* selects the operation op^* with the largest $score$ from I; if the $score$ of op^* is positive, then *HSCA* performs op^* and activates the next search step (Lines 9–14 in Algorithm 4); otherwise, *HSCA* updates the priority values (as described in Section IV-D1) and activates the diversification mode.

E. Diversification Mode

As discussed in Section IV-C, the primary purpose of diversification mode is to diversify the search direction, for exploring promising areas of the search space. In the context of local search, it is recognized that adopting randomized strategy can help explore promising search space. Thus, in the diversification mode, *HSCA* works as follows (Lines 17–20 in Algorithm 4). First, an uncovered tuple τ is randomly selected from U, and a test case σ is randomly selected from E. Then, *HSCA* invokes the constraint solver *ContextSAT* [8] to build a test case σ' covering τ. Finally, *HSCA* performs the operation $op = (\sigma, \sigma')$ on E (*i.e.,* replacing σ with σ' in E).

V. Experimental Design

A. Public Instances

To evaluate *HSCA*, we adopt a benchmarking set of 35 public instances, which is originally collected by Cohen *et al.* [57] and has been studied in the literature [1], [6], [15], [47], [52], [55], [57]–[59]. Among these instances, five are collected from real-world, highly configurable systems, including `apache`, a well-known open-source web server application; `bugzilla`, a prominent bug-tracking system; `gcc`, a widely-used compiler collection; and `spins` and `spinv`, the simulator and verifier of SPIN, a widely-used model checker. The remaining 30 instances are synthetic ones, which are generated to mimic the characteristics of the real-world systems mentioned above. All adopted instances are publicly available in our repository.[1] For more details about these instances, readers can refer to the literature [52], [57], [58].

B. State-of-the-art Competitors

In our experiments, *HSCA* is compared against 4 state-of-the-art CCAG competitors, *i.e., AutoCCAG* [6], *SamplingCA* [8], *CAmpactor* [12], and *PRBOT-its* [47].

AutoCCAG [6] is the current best approach for 4-wise and 5-wise CCAG. Experiments in the literature [6] show that *AutoCCAG* greatly outperforms other well-known algorithms, including *TCA* [1], *CASA* [52] and *CHiP* [14].

SamplingCA [8] is a high-performance CCAG algorithm. As reported in the literature [8], *SamplingCA* performs better than many CCAG algorithms, *e.g., FastCA* [15], [55], *TCA* [1], *CASA* [52], *HHSA* [54] and *ACTS* [44].

CAmpactor [12] is an effective local search CCAG algorithm. According to the literature [12], *CAmpactor* generates smaller CAs than various CCAG algorithms, *e.g., SamplingCA*, *FastCA*, *TCA*, *CASA*, *HHSA*, *ACTS* and *CTLog* [34].

PRBOT-its [47] is a recently-proposed method for building high-strength CAs. Experiments in the literature [47], *PRBOT-its* achieves better performance than existing CCAG algorithms, including *ACTS*, *WCA* [86] and *PBOT-its* [87].

[1]https://github.com/chuanluocs/HSCA

The source code of *AutoCCAG* is provided by its authors [6], while the implementations of *SamplingCA*[2], *CAmpactor*[3] and *PRBOT-its*[4] are publicly available online.

C. Research Questions

In this work, we design our experiments to answer the following research questions (RQs).

RQ1: Can *HSCA* outperform its state-of-the-art competitors when solving 4-wise and 5-wise CCAG?

In this RQ, we compare *HSCA* against its competitors when solving 4-wise and 5-wise CCAG problems.

RQ2: Do the core techniques proposed in this work contribute to the performance improvement of *HSCA*?

In this RQ, we perform ablation study to analyze the contribution of each core technique of *HSCA*.

RQ3: How does the setting of the hyper-parameter impact the practical performance of *HSCA*?

In this RQ, we conduct experiments to study how the setting of *HSCA*'s hyper-parameter (*i.e.,* L) impacts its performance.

RQ4: How does *HSCA* perform when solving the CCAG problem with higher testing strength?

In this RQ, we evaluate the performance of *HSCA* and its competitors when solving the 6-wise CCAG problem.

D. Experimental Setup

In this work, all experiments were performed on a computing machine with AMD EPYC 7763 CPU and 1TB memory, running the operating system of Ubuntu 20.04.4 LTS.

Since *HSCA* and all its competitors are randomized approaches, each algorithm is performed 10 independent runs per instance. Following the standard experimental setup [6], when solving 4-wise CCAG, the cutoff time for each algorithm run is set to 1,000 seconds; when solving 5-wise CCAG, the cutoff time for each algorithm run is set to 10,000 seconds. For *HSCA*, its hyper-parameter L is set to 5,000; we note that the effect of L will be empirically analyzed in Section VI-C. For all competitors of *HSCA*, we adopt the hyper-parameter settings recommended by their authors [6], [8], [12], [47].

Following recent studies [6], [8], [12], for each competing algorithm on each instance, we report the minimum size of the t-wise CAs generated across 10 independent runs, denoted by 'min.', the average size of the t-wise CAs among 10 runs, denoted by 'avg.', and the average running time over 10 runs, denoted by 'time'. All times are measured in seconds. If an algorithm fails to build a t-wise CA for an instance within the cutoff time, we marked the corresponding results with '- -'. For each instance, if a competing algorithm builds the smallest-sized t-wise CA, then its results of 'min.' and 'avg.' are highlighted via **boldface**.

For each instance, we perform Wilcoxon signed-rank tests [88] and calculate the Vargha-Delaney effect sizes [89] for all

[2]https://github.com/chuanluocs/SamplingCA/tree/general
[3]https://github.com/chuanluocs/CAmpactor/tree/general
[4]http://hardlog.udl.cat/static/doc/ctlog/html/index.html

TABLE II
RESULTS OF *HSCA* AND ITS COMPETITORS FOR 4-WISE AND 5-WISE CCAG ON ALL 5 REAL-WORLD INSTANCES AND 5 SYNTHETIC INSTANCES.

Instance	*HSCA*		*AutoCCAG*		*CAmpactor*		*SamplingCA*		*PRBOT-its*	
	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time
$t = 4$										
apache	**734** (**736.5**)	971.7	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
bugzilla	**166** (**166.3**)	444.6	167 (167.6)	390.3	171 (172.9)	867.6	272 (278.2)	50.4	195 (199.6)	601.3
gcc	**340** (**344.3**)	969.5	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
spins	**308** (**308.0**)	105.7	**308** (310.0)	81.2	344 (356.9)	354.0	454 (462.2)	7.2	390 (399.2)	963.6
spinv	**1,091** (**1,094.2**)	863.2	1,107 (1,111.4)	760.0	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_2	**624** (**628.2**)	920.9	676 (680.7)	950.7	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_4	**275** (**276.6**)	654.8	298 (299.3)	637.0	399 (405.6)	991.6	477 (482.6)	217.4	342 (348.3)	785.8
Syn_9	**200** (**200.0**)	635.9	**200** (**200.0**)	222.3	209 (211.5)	977.6	329 (334.9)	142.7	237 (242.0)	879.5
Syn_12	**1,257** (**1,262.5**)	967.6	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_18	**1,840** (**1,845.5**)	992.7	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
$t = 5$										
apache	**3,643** (**3,702.8**)	9,996.6	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
bugzilla	**555** (**558.6**)	6,747.6	559 (559.6)	7,905.7	741 (750.2)	9,947.6	856 (860.6)	2,467.5	660 (673.7)	9,452.1
gcc	**1,403** (**1,427.4**)	9,983.7	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
spins	**1,174** (**1,174.0**)	184.1	**1,174** (**1,174.0**)	487.9	1,223 (1,232.3)	9,831.9	1,519 (1,533.4)	198.4	1,294 (1,306.7)	9,710.6
spinv	**5,577** (**5,586.2**)	9,691.7	5,796 (5,847.0)	9,974.5	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_2	**2,810** (**2,817.7**)	9,530.3	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_4	**984** (**990.2**)	9,765.2	1,022 (1,025.7)	8,576.4	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_9	**662** (**667.0**)	8,556.7	665 (667.4)	7,720.6	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_12	**6,332** (**6,360.6**)	9,988.4	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
Syn_18	**10,821** (**11,041.0**)	9,998.3	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -

pairwise comparisons between *HSCA* and each of its competitors. Specifically, the following criteria are considered: a) all p-values of the Wilcoxon signed-rank tests at a 95% confidence level are less than 0.05; b) the Vargha-Delaney effect sizes for all pairwise comparisons are greater than 0.71, which suggests a significant effect size [6]–[8], [12], [89], [90]. If both criteria are met, we concluded that the performance improvement of *HSCA* over its comparative algorithms on the corresponding instance is statistically significant and meaningful, and the results of *HSCA* are highlighted via underline.

VI. EXPERIMENTAL RESULTS

A. RQ1: Comparison with State of the Art

We compare *HSCA* with all its competitors for 4-wise and 5-wise CCAG on all 35 instances. Table II presents the results of 5 real-world instances and 5 randomly selected, synthetic instances. The complete results on all instances are publicly available in our repository.[1] According to Table II, *HSCA* generates 4-wise and 5-wise CAs of notably smaller size than all its competitors. More encouragingly, among 35 instances, *HSCA* can successfully build 4-wise and 5-wise CAs for 35 and 29 instances, respectively, including 11 and 15 instances that all its competitors fail. Our results show that *HSCA* greatly advances the state of the art in 4-wise and 5-wise CCAG.

B. RQ2: Effectiveness of Each Core Technique

HSCA proposes three novel and core techniques, *i.e.,* multi-round CA generation mechanism (Section IV-B2), dynamic priority assigning technique (Section IV-D1), and variable grouping strategy (Section IV-D2). To study the effect of all core techniques, based on *HSCA* we develop three alternative versions *Alt-1*, *Alt-2* and *Alt-3*. *Alt-1* is *HSCA*'s alternative version that replaces the multi-round CA generation mechanism with *AutoCCAG* [6] in the generation stage, since *AutoCCAG* is the current best approach for solving 4-wise and 5-wise CCAG (also confirmed in Table II). *Alt-2* is *HSCA*'s alternative versions that directly works without the dynamic priority assigning technique. *Alt-3* is *HSCA*'s alternative version that replaces the variable grouping strategy with the existing constraint solver-based method [8], [12] (as described in Section IV-D2) when handling additional clauses.

TABLE III
RESULTS OF *HSCA* AND ITS ALTERNATIVE VERSIONS.

Instance	*HSCA*	*Alt-1*	*Alt-2*	*Alt-3*
	min. (avg.) time	min. (avg.) time	min. (avg.) time	min. (avg.) time
$t = 4$:				
apache	**734** (**736.5**) 971.7	- - (- -) - -	772 (783.5) 941.1	1,101 (1,106.0) 927.1
bugzilla	**166** (**166.3**) 444.6	**166** (166.6) 775.7	**166** (166.6) 643.6	172 (174.8) 520.1
gcc	**340** (**344.3**) 969.5	- - (- -) - -	363 (369.3) 894.9	414 (422.2) 643.0
spins	**308** (**308.0**) 105.7	**308** (**308.0**) 133.6	310 (314.9) 425.5	387 (400.5) 255.3
spinv	**1,091** (**1,094.2**) 863.2	1,099 (1,101.6) 987.1	1,103 (1,105.1) 820.3	1,469 (1,482.3) 302.5
$t = 5$:				
apache	**3,643** (**3,702.8**) 9,996.6	- - (- -) - -	4,070 (4,402.4) 9,973.9	4,675 (4,726.7) 9,948.9
bugzilla	**555** (558.6) 6,747.6	558 (558.7) 8,507.2	556 (**557.4**) 7,518.2	570 (590.5) 8,853.1
gcc	**1,403** (**1,427.4**) 9,983.7	- - (- -) - -	1,525 (1,621.1) 9,991.3	1,583 (1,615.6) 9,964.1
spins	**1,174** (**1,174.0**) 184.1	**1,174** (**1,174.0**) 633.9	**1,174** (**1,174.0**) 1,243.6	1,200 (1,205.0) 6,976.2
spinv	**5,577** (**5,586.2**) 9,691.7	5,783 (5,811.0) 9,978.4	5,692 (5,715.3) 9,930.8	7,143 (7,158.5) 6,695.0

Due to page limit, Table III reports the results of *HSCA* and all its alternative versions on 5 real-world instances, with complete results publicly available in our repository.[1] According to Table III, *HSCA* performs much better than all its alternative versions, indicating that each core technique contributes to the performance improvement of *HSCA*.

TABLE IV
RESULTS OF *HSCA* WITH DIFFERENT SETTINGS OF L.

Instance	$L = 500$	$L = 5,000$	$L = 50,000$
	min. (avg.) time	min. (avg.) time	min. (avg.) time
$t = 4$:			
apache	759 (767.8) 956.8	**734** (**736.5**) 971.7	**734** (738.1) 956.8
bugzilla	**166** (166.4) 810.4	**166** (**166.3**) 444.6	**166** (166.9) 810.4
gcc	357 (368.9) 954.1	**340** (**344.3**) 969.5	342 (345.3) 954.1
spins	**308** (308.7) 147.2	**308** (**308.0**) 105.7	**308** (309.4) 147.2
spinv	1,092 (1,094.6) 871.5	**1,091** (**1,094.2**) 863.2	1,114 (1,134.2) 871.5
$t = 5$:			
apache	3,777 (4,164.5) 9,996.0	**3,643** (**3,702.8**) 9,996.6	3,644 (3,868.2) 9,996.0
bugzilla	556 (559.4) 7,938.9	**555** (**558.6**) 6,747.6	557 (559.0) 7,938.9
gcc	1,435 (1,611.2) 9,984.3	**1,403** (**1,427.4**) 9,983.7	1,433 (1,452.8) 9,984.3
spins	**1,174** (**1,174.0**) 232.7	**1,174** (**1,174.0**) 184.1	**1,174** (**1,174.0**) 232.7
spinv	**5,570** (**5,578.8**) 9,457.6	5,577 (5,586.2) 9,691.7	5,711 (5,726.6) 9,457.6

C. RQ3: Impact of Hyper-parameter Setting

HSCA has one hyper-parameter L, and its value decides when *HSCA*'s first pass of optimization terminates (as described in Section IV-C). Table IV reports the performance of *HSCA* with different settings of L (*i.e.,* $L = 500$, $5,000$ and $50,000$). To save space, Table IV presents the results on 5 real-world instances, and the full results on all instances are publicly available in our repository.[1] As can be observed from Table IV, *HSCA* achieves competitive performance across different settings of L. Hence, our results indicate that *HSCA* is not sensitive to the setting of L, demonstrating its robustness.

D. RQ4: Performance of HSCA for Solving 6-wise CCAG

Here we perform experiments to study the performance of *HSCA* when solving the 6-wise CCAG problem. The comparative results of *HSCA* and all its competitors are reported in Table V, where the cutoff time for each algorithm run is set to 10,000 seconds. Due to page limit, Table V reports the comparative results on 5 real-world instances, and the complete results on all instances are publicly available in our repository.[1] According to Table V, there are two real-world instances (*i.e.,* bugzilla and spins), where at least one of *HSCA*'s competitors can generate 6-wise CA. On these two real-world instances, *HSCA* can generate 6-wise CAs of significantly smaller sizes than its competitors. More encouragingly, *HSCA* successfully generate 6-wise CA for the spinv instance, while all its competitors fail to build 6-wise CA for this instance. Our results demonstrate the superiority of *HSCA* when solving the high-strength CCAG problem.

VII. DISCUSSIONS

In this section, we evaluate *HSCA* on 2-wise and 3-wise CCAG. Then, we analyze its fault detection capability. Finally, we discuss the treats to the validity of this work.

A. Evaluation on 2-wise and 3-wise CCAG

In addition to evaluating *HSCA* on high-strength CCAG (*e.g.,* 4-wise and 5-wise CCAG), we conducted experiments to assess its performance on 2-wise and 3-wise CCAG. For the evaluation on 3-wise CCAG, besides the competitors described in Section V-B, we additionally included another CCAG approach called *ScalableCA* [91], which is specifically designed for 3-wise CCAG and is the current best approach for 3-wise CCAG. Due to page limit, the comparative results are publicly available in our repository.[1] Out of all 35 instances, *HSCA* achieves the best performance on 35 and 34 instances when solving 2-wise and 3-wise CCAG, respectively.

B. Analysis of Fault Detection Capability

As discussed in Section III-B, existing CCAG algorithms suffer from severe high-strength challenge due to the existence of huge numbers of t-wise tuples. Compared to CCAG algorithms, there exists a class of approximate methods [92]. These approximate methods are designed to produce test suites that achieve high t-wise coverage; however, they do not guarantee the full coverage of all possible valid t-wise tuples. To demonstrate the practical importance of building CAs, we empirically evaluate the fault detection capability of *HSCA* and a well-known approximate method called *PLEDGE* [92], whose implementation is publicly available online.[5]

For this purpose, we employ 9 real-world, highly configurable systems as our subjects. These subjects were originally collected and utilized in a recent study [25]. In this evaluation, we use fault detection rate as the evaluation metric. For a given subject and a test suite T, the fault detection rate of T is defined as the fraction between the number of faults, which can be detected by T, and the total number of faults within the given subject. Since *PLEDGE* [92] generates the test suite of a target size, rather than building t-wise CAs, in this evaluation, for each subject we first use *HSCA* to build 4-wise and 5-wise CAs, and then we employ *PLEDGE* to generate test suites of two particular sizes (*i.e.,* the sizes of *HSCA*'s 4-wise and 5-wise CAs). The average fault detection rate achieved by *HSCA* and *PLEDGE* across all 9 subjects is shown in Table VI. From Table VI, *HSCA* achieves higher fault detection rate than *PLEDGE*, indicating the practical value of *HSCA* in real-world applications.

C. Threats to Validity

Representativeness of the Instances. The instances adopted in this work include real-world instances and synthetic ones. As described in Section V-A, these adopted instances have been extensively studied in recent literature [1], [6], [15], [47], [55]. Also, these adopted instances are of general scenario (*i.e.,* each option can take multiple possible values). We note that prior studies [8], [12], [91] employ a different set of instances, where all instances are of binary scenario (*i.e.,* each option can take only 2 possible values). Since *HSCA* is a CCAG algorithm that handles the general scenario, we adopted

[5]https://henard.net/research/SPL/TSE_2014/

TABLE V
RESULTS OF *HSCA* AND ITS COMPETITORS FOR 6-WISE CCAG ON 5 REAL-WORLD INSTANCES.

Instance	*HSCA*		*AutoCCAG*		*CAmpactor*		*SamplingCA*		*PRBOT-its*	
	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time	min. (avg.)	time
apache	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
bugzilla	**1,780** (**1,812.9**)	9,648.7	2,183 (2,183.0)	8,855.4	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
gcc	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -
spins	**3,060** (**3,064.9**)	5,986.8	3,160 (3,167.7)	9,092.3	4,406 (4,426.9)	9,953.0	4,465 (4,486.2)	4,449.6	- - (- -)	- -
spinv	**32,275** (**32,307.8**)	9,152.7	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -	- - (- -)	- -

TABLE VI
AVERAGE FAULT DETECTION RATE ACHIEVED BY *HSCA* AND *PLEDGE*.

	$t=4$	$t=5$
HSCA's average fault detection rate	97.2%	100.0%
PLEDGE's average fault detection rate	91.5%	97.1%

those instances of general scenario in this work. Thus, this potential threat can be alleviated.

Randomness of Algorithms. In our experiments, all algorithms are randomized, making a single run per instance insufficient for precise performance evaluation. To address this, we conducted 10 independent runs for each algorithm per instance, following recent studies [6], [8], [12]. Also, we have performed significance tests and calculated effect sizes to analyze the results, thereby mitigating this potential issue.

VIII. RELATED WORK

Combinatorial interaction testing (CIT) is a key area in software testing, with constrained covering array generation (CCAG) as its core problem [19], [93]–[95]. Extensive studies show that higher-strength covering array (CA) has stronger fault detection ability [20]–[24], making it crucial to solve high-strength CCAG problem (*e.g.,* 4-wise and 5-wise CCAG).

Practical CCAG algorithms can be divided into three major categories, *i.e.,* constraint-encoding algorithms (*e.g.,* [30]–[34]), greedy algorithms (*e.g.,* [8], [18], [35]–[47]), and meta-heuristic algorithms (*e.g.,* [1], [6], [12], [15], [22], [36], [48]–[56]). Constraint-encoding algorithms encode CCAG into other combinatorial optimization problems, but they can only generate low-strength CAs [30]–[34]. Greedy algorithms, subdivided into one-test-at-a-time (OTAT) paradigm [35]–[40] and in-parameter-order (IPO) paradigm [18], [41]–[47], usually generate large CAs in practice, which is not applicable for real-world scenarios with limited testing budget. Compared to constraint-encoding algorithms and greedy algorithms, it is recognized that meta-heuristic algorithms can generate smaller CAs but require longer running time [1], [12], [15], [52], [55]. However, existing CCAG algorithms face a significant high-strength challenge. They typically require a substantial amount of time to build large 4-wise and 5-wise CAs, and often fail to produce these CAs within a reasonable time [6], [47].

There are very few studies focusing on 4-wise and 5-wise CCAG problems. *AutoCCAG* [6] is currently the best approach for solving 4-wise and 5-wise CCAG, while *PRBOT-its* [47] is a recent method addressing high-strength CCAG problems. However, as shown in our experiments in Section VI, both *AutoCCAG* and *PRBOT-its* fail to generate 4-wise and 5-wise CAs for many instances. A recent work [96] focuses on generating high-strength CAs for configurable systems without constraints. Unfortunately, the proposed method [96] cannot handle constraints, making it inapplicable to the CCAG problem studied in this work. Hence, it is crucial to design effective algorithms for solving 4-wise and 5-wise problems.

This work introduces *HSCA*, a novel algorithm that exhibits the effectiveness when solving 4-wise and 5-wise CCAG. Notably, *HSCA* can produce 4-wise and 5-wise CAs for many instances where existing algorithms fail, confirming its capability to mitigate the high-strength challenge.

IX. CONCLUSION

In this work, we introduce *HSCA*, a novel and effective CCAG algorithm that significantly alleviates high-strength challenges. *HSCA* incorporates three innovative techniques, *i.e.,* multi-round CA generation mechanism, dynamic priority assigning technique, and variable grouping strategy, to enhance performance. Extensive experiments on 35 real-world and synthetic instances show that *HSCA* generates much smaller 4-wise and 5-wise CAs than existing CCAG algorithms. Notably, among 35 instances, *HSCA* can build 4-wise and 5-wise CAs for 35 and 29 instances, respectively, including 11 and 15 instances where existing algorithms fail, showing its ability to mitigate the severe high-strength challenge.

Data Availability: The implementation of *HSCA*, all adopted instances and detailed results are publicly available in our repository: **https://github.com/chuanluocs/HSCA**.

ACKNOWLEDGMENT

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB3307503, in part by the National Natural Science Foundation of China under Grants 62202025 and 62302528, in part by Beijing Natural Science Foundation under Grant L241050, in part by the Young Elite Scientist Sponsorship Program by CAST under Grant YESS20230566, in part by CCF-Huawei Populus Grove Fund under Grant CCF-HuaweiFM2024005, and in part by the Fundamental Research Fund Project of Beihang University.

REFERENCES

[1] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "TCA: An efficient two-mode meta-heuristic algorithm for combinatorial test generation," in *Proceedings of ASE 2015*, 2015, pp. 494–505.

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

[3] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based sampling of software configuration spaces," in *Proceedings of ICSE 2019*, 2019, pp. 1084–1094.

[4] E. Baranov, A. Legay, and K. S. Meel, "Baital: An adaptive weighted sampling approach for improved t-wise coverage," in *Proceedings of ESEC/FSE 2020*, 2020, pp. 1114–1126.

[5] M. Park, H. Jang, T. Byun, and Y. Choi, "Property-based testing for LG home appliances using accelerated software-in-the-loop simulation," in *Proceedings of ICSE-SEIP 2020*, 2020, pp. 120–129.

[6] C. Luo, J. Lin, S. Cai, X. Chen, B. He, B. Qiao, P. Zhao, Q. Lin, H. Zhang, W. Wu, S. Rajmohan, and D. Zhang, "AutoCCAG: An automated approach to constrained covering array generation," in *Proceedings of ICSE 2021*, 2021, pp. 201–212.

[7] C. Luo, B. Sun, B. Qiao, J. Chen, H. Zhang, J. Lin, Q. Lin, and D. Zhang, "LS-Sampling: An effective local search based sampling approach for achieving high t-wise coverage," in *Proceedings of ESEC/FSE 2021*, 2021, pp. 1081–1092.

[8] 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 ESEC/FSE 2022*, 2022, pp. 1185–1197.

[9] C. Luo, J. Song, Q. Zhao, Y. Li, S. Cai, and C. Hu, "Generating pairwise covering arrays for highly configurable software systems," in *Proceedings of SPLC 2023*, 2023, pp. 261–267.

[10] S. Mühlbauer, F. Sattler, C. Kaltenecker, J. Dorn, S. Apel, and N. Siegmund, "Analysing the impact of workloads on modeling the performance of configurable software systems," in *Proceedings of ICSE 2023*, 2023, pp. 2085–2097.

[11] M. Weber, C. Kaltenecker, F. Sattler, S. Apel, and N. Siegmund, "Twins or false friends? a study on energy consumption and performance of configurable software," in *Proceedings of ICSE 2023*, 2023, pp. 2098–2110.

[12] Q. Zhao, C. Luo, S. Cai, W. Wu, J. Lin, H. Zhang, and C. Hu, "CAmpactor: A novel and effective local search algorithm for optimizing pairwise covering arrays," in *Proceedings of ESEC/FSE 2023*, 2023, pp. 81–93.

[13] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[14] H. Mercan, C. Yilmaz, and K. Kaya, "CHiP: A configurable hybrid parallel covering array constructor," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1270–1291, 2019.

[15] J. Lin, S. Cai, C. Luo, Q. Lin, and H. Zhang, "Towards more efficient meta-heuristic algorithms for combinatorial test generation," in *Proceedings of ESEC/FSE 2019*, 2019, pp. 212–222.

[16] J. Petke, S. Yoo, M. B. Cohen, and M. Harman, "Efficiency and early fault detection with lower and higher strength combinatorial interaction testing," in *Proceedings of ESEC/FSE 2013*, 2013, pp. 26–36.

[17] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "Comparative analysis of constraint handling techniques for constrained combinatorial testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2549–2562, 2021.

[18] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *Proceedings of HASE 1998*, 1998, pp. 254–261.

[19] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, vol. 43, no. 2, pp. 11:1–11:29, 2011.

[20] R. Kuhn, R. N. Kacker, J. Y. Lei, and D. E. Simos, "Input space coverage matters," *Computer*, vol. 53, no. 1, pp. 37–44, 2020.

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

[22] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, "A variable strength interaction testing of components," in *Proceedings of COMPAC 2003*, 2003, pp. 413–418.

[23] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *Proceedings of SEW 2006*, 2006, pp. 153–158.

[24] R. Huang, H. Chen, Y. Zhou, T. Yueh Chen, D. Towey, M. Fai Lau, S. Ng, R. Merkel, and J. Chen, "Covering Array Constructors: An Experimental Analysis of Their Interaction Coverage and Fault Detection," *The Computer Journal*, vol. 64, no. 5, pp. 762–788, 04 2020.

[25] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "An empirical comparison of combinatorial testing, random testing and adaptive random testing," *IEEE Transactions on Software Engineering*, vol. 46, no. 3, pp. 302–320, 2020.

[26] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of ICSE 1994*, 1994, pp. 191–200.

[27] G. Rothermel and M. Harrold, "Empirical studies of a safe regression test selection technique," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 401–419, 1998.

[28] V. Okun, P. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," *WSEAS Transactions on Systems*, vol. 2, pp. 77–82, 12 2002.

[29] M. Park, H. Jang, T. Byun, and Y. Choi, "Property-based testing for lg home appliances using accelerated software-in-the-loop simulation," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 120–129.

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

[31] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue, "Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers," in *Proceedings of LPAR 2010*, 2010, pp. 112–126.

[32] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, pp. 191–207, 2014.

[33] A. Yamada, T. Kitamura, C. Artho, E. Choi, Y. Oiwa, and A. Biere, "Optimization of combinatorial testing by incremental SAT solving," in *Proceedings of ICST 2015*, 2015, pp. 1–10.

[34] C. Ansótegui, F. Manyà, J. Ojeda, J. M. Salvia, and E. Torres, "Incomplete MaxSAT approaches for combinatorial testing," *Journal of Heuristics*, vol. 28, no. 4, pp. 377–431, 2022.

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

[36] 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.

[37] ——, "A density-based greedy algorithm for higher strength covering arrays," *Software Testing, Verification and Reliability*, vol. 19, no. 1, pp. 37–53, 2009.

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

[39] A. Yamada, A. Biere, C. Artho, T. Kitamura, and E. Choi, "Greedy combinatorial test case generation using unsatisfiable cores," in *Proceedings of ASE 2016*, 2016, pp. 614–624.

[40] R. C. Bryce, C. J. Colbourn, and M. B. Cohen, "A framework of greedy methods for constructing interaction test suites," in *Proceedings of ICSE 2005*, 2005, pp. 146–155.

[41] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A general strategy for t-way software testing," in *Proceedings of ECBS 2007*, 2007, pp. 549–556.

[42] ——, "IPOG/IPOG-D: Efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[43] Z. Wang, C. Nie, and B. Xu, "Generating combinatorial test suite for interaction relationship," in *Proceedings of SOQUA 2007*, 2007, pp. 55–61.

[44] L. Yu, Y. Lei, M. N. Borazjany, R. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *Proceedings of ICST 2013*, 2013, pp. 242–251.

[45] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich, "YASA: yet another sampling algorithm," in *Proceedings of VaMoS 2020*, 2020, pp. 4:1–4:10.

[46] L. Kampel, B. Garn, and D. E. Simos, "Combinatorial methods for modelling composed software systems," in *Proceedings of ICSTW 2017*, 2017, pp. 229–238.

[47] C. Ansótegui and E. Torres, "Effectively computing high strength mixed covering arrays with constraints," *Journal of Parallel and Distributed Computing*, vol. 185, p. 104791, 2024.

[48] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings ICSE 2003*, 2003, pp. 38–48.

[49] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Augmenting simulated annealing to build interaction test suites," in *Proceedings of ISSRE 2003*, 2003, pp. 394–405.

[50] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *Proceedings of CEC 2003*, 2003, pp. 1420–1424.

[51] J. D. McCaffrey, "Generation of pairwise test sets using a genetic algorithm," in *Proceedings of COMPSAC 2009*, 2009, pp. 626–631.

[52] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Proceedings of International Symposium on Search Based Software Engineering 2009*, 2009, pp. 13–22.

[53] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *Proceedings of COCOA 2010*, 2010, pp. 51–64.

[54] Y. Jia, M. B. Cohen, M. Harman, and J. Petke, "Learning combinatorial interaction test generation strategies using hyperheuristic search," in *Proceedings of ICSE 2015*, 2015, pp. 540–550.

[55] J. Lin, S. Cai, B. He, Y. Fu, C. Luo, and Q. Lin, "FastCA: An effective and efficient tool for combinatorial covering array generation," in *Proceedings of ICSE 2021 (Companion Volume)*, 2021, pp. 77–80.

[56] Y. Xiang, H. Huang, S. Li, M. Li, C. Luo, and X. Yang, "Automated test suite generation for software product lines based on quality-diversity optimization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 46:1–46:52, 2024.

[57] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of ISSTA 2007*, 2007, pp. 129–139.

[58] ——, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[59] 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, no. 1, pp. 61–102, 2011.

[60] J. Oh, P. Gazzillo, and D. S. Batory, "*t*-wise coverage by uniform sampling," in *Proceedings of SPLC 2019*, 2019, pp. 15:1–15:4.

[61] 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.

[62] M. Mendonça, A. Wasowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *Proceedings of SPLC 2009*, 2009, pp. 231–240.

[63] J. Sun, H. Zhang, Y. Li, and H. H. Wang, "Formal semantics and verification for feature modeling," in *Proceedings of ICECCS 2005*, 2005, pp. 303–312.

[64] D. S. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of SPLC 2005*, 2005, pp. 7–20.

[65] S. Apel, D. S. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[66] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of ICSE 2016*, 2016, pp. 643–654.

[67] S. D. Prestwich, "CNF encodings," in *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2021, vol. 336, pp. 75–100.

[68] C. Henard, M. Papadakis, and Y. L. Traon, "Flattening or not of the combinatorial interaction testing models?" in *Proceedings of ICST Workshops 2015*, 2015, pp. 1–4.

[69] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability - Second Edition*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, vol. 336.

[70] S. Cai, K. Su, C. Luo, and A. Sattar, "NuMVC: An efficient local search algorithm for minimum vertex cover," *Journal of Artificial Intelligence Research*, vol. 46, pp. 687–716, 2013.

[71] X. Cai, H. Sun, Q. Zhang, and Y. Huang, "A grid weighted sum pareto local search for combinatorial multi and many-objective optimization," *IEEE Transactions on Cybernetics*, vol. 49, no. 9, pp. 3586–3598, 2019.

[72] H. H. Hoos and T. Stützle, *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[73] C. Luo, S. Cai, W. Wu, Z. Jie, and K. Su, "CCLS: An efficient local search algorithm for weighted maximum satisfiability," *IEEE Transactions on Computers*, vol. 64, no. 7, pp. 1830–1843, 2015.

[74] C. Luo, S. Cai, K. Su, and W. Huang, "CCEHC: An efficient local search algorithm for weighted partial maximum satisfiability," *Artificial Intelligence*, vol. 243, pp. 26–44, 2017.

[75] C. Luo, H. H. Hoos, S. Cai, Q. Lin, H. Zhang, and D. Zhang, "Local search with efficient automatic configuration for minimum vertex cover," in *Proceedings of IJCAI 2019*, 2019, pp. 1297–1304.

[76] C. Luo, B. Qiao, X. Chen, P. Zhao, R. Yao, H. Zhang, W. Wu, A. Zhou, and Q. Lin, "Intelligent virtual machine provisioning in cloud computing," in *Proceedings of IJCAI 2020*, 2020, pp. 1495–1502.

[77] M. Mavrovouniotis, F. M. Müller, and S. Yang, "Ant colony optimization with local search for dynamic traveling salesman problems," *IEEE Transactions on Cybernetics*, vol. 47, no. 7, pp. 1743–1756, 2017.

[78] J. Shi, Q. Zhang, and J. Sun, "PPLS/D: Parallel pareto local search based on decomposition," *IEEE Transactions on Cybernetics*, vol. 50, no. 3, pp. 1060–1071, 2020.

[79] Y. Chu, S. Cai, and C. Luo, "NuWLS: Improving local search for (weighted) partial MaxSAT by new weighting techniques," in *Proceedings of AAAI 2023*, 2023, pp. 3915–3923.

[80] C. Luo, W. Xing, S. Cai, and C. Hu, "NuSC: An effective local search algorithm for solving the set covering problem," *IEEE Transactions on Cybernetics*, vol. 54, no. 3, pp. 1403–1416, 2024.

[81] C. Liu, G. Liu, C. Luo, S. Cai, Z. Lei, W. Zhang, Y. Chu, and G. Zhang, "Optimizing local search-based partial MaxSAT solving via initial assignment prediction," *Science China Information Sciences*, vol. 68, no. 2, pp. 122 101:1–122 101:15, 2025.

[82] E. Baranov, S. Chakraborty, A. Legay, K. S. Meel, and N. V. Vinodchandran, "A scalable *t*-wise coverage estimator: Algorithms and applications," *IEEE Transactions on Software Engineering*, 2024.

[83] C. M. Li and W. Huang, "Diversification and determinism in local search for satisfiability," in *Proceedings of SAT 2005*, 2005, pp. 158–172.

[84] C. Luo, S. Cai, K. Su, and W. Wu, "Clause states based configuration checking in local search for satisfiability," *IEEE Transactions on Cybernetics*, vol. 45, no. 5, pp. 1014–1027, 2015.

[85] W. Michiels, E. H. L. Aarts, and J. H. M. Korst, *Theoretical aspects of local search*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2007.

[86] Y. Fu, Z. Lei, S. Cai, J. Lin, and H. Wang, "WCA: A weighting local search for constrained combinatorial test optimization," *Information and Software Technology*, vol. 122, p. 106288, 2020.

[87] C. Ansótegui, J. Ojeda, and E. Torres, "Building high strength mixed covering arrays with constraints," in *Proceedings of CP 2021*, 2021, pp. 12:1–12:17.

[88] W. J. Conover, *Practical Nonparametric Statistics*. Conover, 1999.

[89] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[90] F. Sarro, M. Harman, Y. Jia, and Y. Zhang, "Customer rating reactions can be predicted purely using app features," in *Proceedings of RE 2018*, 2018, pp. 76–87.

[91] C. Luo, S. Lyu, Q. Zhao, W. Wu, H. Zhang, and C. Hu, "Beyond pairwise testing: Advancing 3-wise combinatorial interaction testing for highly configurable systems," in *Proceedings of ISSTA 2024*, 2024, pp. 641–653.

[92] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon, "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.

[93] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 6:1–6:45, 2014.

[94] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*. CRC press, 2013.

[95] J. Zhang, Z. Zhang, and F. Ma, *Automatic Generation of Combinatorial Test Data*. Springer, 2014.

[96] X. Guo, X. Song, J. Zhou, F. Wang, and K. Tang, "An effective approach to high strength covering array generation in combinatorial testing," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4566–4593, 2023.