Beyond Pairwise Testing: Advancing 3-wise Combinatorial Interaction Testing for Highly Configurable Systems

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

Shuangyu Lyu
Beihang University
Beijing, China
19374290@buaa.edu.cn

Qiyuan Zhao
National University of Singapore
Singapore, Singapore
qiyuanz@comp.nus.edu.sg

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

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

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

Abstract

To meet the rising demand for software customization, highly configurable software systems play key roles in practice. Combinatorial interaction testing (CIT) is recognized as an effective approach for testing such systems. For CIT, the most important problem is constrained covering array generation (CCAG), which aims to construct a minimum-sized t-wise covering array (CA), where t denotes testing strength. Compared to pairwise testing (*i.e.,* 2-wise CIT) that is a widely-used CIT technique, 3-wise CIT can discover more faults and bring more benefit in real-world applications. However, current state-of-the-art CCAG algorithms suffer from the severe scalability challenge for 3-wise CIT, which renders them ineffective in building 3-wise CAs for highly configurable systems. In this work, we perform an empirical study on various practical, highly configurable systems to present that it is promising to build 3-wise CA through extending 2-wise CA. Inspired by this, we propose *ScalableCA*, a novel and scalable algorithm that can effectively alleviate the scalability challenge for 3-wise CIT. Further, *ScalableCA* introduces three new and effective techniques, including fast invalidity detection, uncovering-guided sampling, and remainder-aware local search, to enhance its performance. Our experiments on extensive real-world, highly configurable systems show that, compared to current state-of-the-art algorithms, *ScalableCA* requires one to two orders of magnitude less running time to build 3-wise CA of 38.9% smaller size in average for large-scale instances. Our results indicate that *ScalableCA* greatly advances the state of the art in 3-wise CIT.

CCS Concepts

• Software and its engineering → Software testing and debugging; • Theory of computation → Randomized local search.

*Corresponding author.

Keywords

Combinatorial Interaction Testing, Satisfiability, Sampling

ACM Reference Format:
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 (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680309

1 Introduction

Highly configurable software systems, which are crucial for satisfying user demands, pose a testing challenge due to the exponential growth in possible configurations with the number of options [4, 28, 44, 52, 59, 61, 67, 71, 72, 86, 97]. Recent studies indicate that such systems typically offer around one thousand configurable options, rendering exhaustive testing impractical [4, 59, 61, 97].

Combinatorial interaction testing (CIT) is a suitable approach for testing these systems [4, 59, 61, 97]. CIT aims to build a test suite of acceptable size (*i.e.,* a set of a reasonable number of configurations) to reveal the faults triggered by the combinations of any t options, where t is a small integer denoting testing strength [44, 52, 65, 93]. For configurable systems, a t-wise tuple is a key concept and denotes a combination of the values of t options. The target of t-wise CIT is to form a t-wise covering array (CA), covering all t-wise tuples while minimizing its size to reduce testing cost [43, 44, 52]. Further, real-world configurable systems usually present hard constraints, like exclusiveness and dependencies, on option interactions [73, 88]. Ensuring that each test case in t-wise CA satisfies all constraints is crucial to avoid faulty outcomes and wasted resources [61, 88, 97]. A test case is valid if it satisfies all constraints; also, a t-wise tuple is valid if it is covered by at least one valid test case. The problem of t-wise constrained covering array generation (CCAG) is to form a minimum-sized t-wise CA with only valid test cases. Effectively solving CCAG remains a big challenge [39, 43, 44, 52, 68].

With the increment in the value of t, the t-wise CCAG problem becomes significantly more difficult [42–44, 52]. While pairwise testing (*i.e.,* 2-wise CIT) is prevalent due to its cost-effectiveness [61, 97], empirical studies on extensive real-world, highly configurable systems show that pairwise testing only detects roughly 77% of faults, whereas 3-wise CIT identifies over 95% of faults [33–36]. Given this, a deeper focus on 3-wise CIT is imperative.

Current CCAG algorithms mainly fall into four categories, *i.e.,* constraint-encoding algorithms (*e.g.,* [1, 3, 24, 92, 96]), greedy algorithms (*e.g.,* [8–10, 12, 81, 91]), incremental generation algorithms (*e.g.,* [29, 31, 37–39, 82, 83, 85, 94]), and meta-heuristic algorithms (*e.g.,* [8, 13–15, 18, 20, 21, 27, 43, 44, 52, 62, 90]). However, existing CCAG algorithms suffer from the severe scalability challenge; that is, they cannot effectively handle large-scale CCAG instances [61, 74, 89, 97]. For example, recent studies [61, 97] indicate that existing algorithms struggle with large-scale 2-wise CCAG instances (*e.g.,* highly configurable systems with around one thousand options), taking extensive time and producing large test suites, which degrades both efficiency and effectiveness of the testing process.

Key Observation. Two recent algorithms, *SamplingCA* [61] and *CAmpactor* [97], have addressed the scalability challenge for pairwise testing and advanced the state of the art in efficiently constucting small 2-wise CAs even for systems with over a thousand options. However, beyond pairwise testing, the scalability challenge remains for 3-wise CIT due to the vast number of valid 3-wise tuples; our experiments (Section 6) show that both *SamplingCA* and *CAmpactor* cannot generate 3-wise CAs effectively and efficiently for various highly configurable systems. Despite this deficiency, from our empirical study (Section 3.2) on extensive highly configurable systems, we observe that 2-wise CA covers most valid 3-wise tuples, which motivates us to build 3-wise CA based on 2-wise CA.

Based on this key observation, we develop *ScalableCA*, a novel and scalable algorithm that can alleviate the scalability challenge for 3-wise CIT. When solving the 3-wise CCAG problem, *ScalableCA* consists of three stages, *i.e.,* initialization stage, sampling stage, and optimization stage. In the initialization stage, *ScalableCA* builds a 2-wise CA A and then obtains the collection U of all remaining, valid 3-wise tuples not covered by A. In the sampling stage, *ScalableCA* builds another test suite T via a new and effective sampling method, for covering all valid 3-wise tuples in U. In the optimization stage, *ScalableCA* reduces the size of T while preserving all valid 3-wise tuples in U being covered by T. That is, the optimization stage finds a test suite T containing a small number of test cases, such that all valid 3-wise tuples in U are covered. After all stages are performed, the union of A and T (*i.e.,* $A \cup T$) is a 3-wise CA and is the output of *ScalableCA*, since $A \cup T$ covers all valid 3-wise tuples.

Compared to current state-of-the-art algorithms, the major advantages of *ScalableCA* are as follows. First, different from invoking a costly process to directly build the 3-wise CA, in the initialization stage *ScalableCA* generates a 2-wise CA of reasonable size, covering the majority of valid 3-wise tuples. Since building a 2-wise CA is fast, the efficiency of *ScalableCA* is enhanced; meanwhile, thanks to the high 3-wise coverage of a 2-wise CA, the problem space (*i.e.,* the number of uncovered, valid 3-wise tuples) is significantly reduced. Second, rather than dealing with the universal set of all valid 3-wise tuples, in both sampling and optimization stages *ScalableCA* targets to generate another test suite of minimum size to cover all remaining valid 3-wise tuples. Through this way, *ScalableCA* operates on a limited number of remaining, valid 3-wise tuples, so the effectiveness of *ScalableCA* can be greatly improved. Moreover, we propose novel algorithmic techniques, *i.e.,* fast invalidity detection, uncovering-guided sampling, and remainder-aware local search, to strengthen the performance of *ScalableCA*.

Extensive experiments on large-scale instances present that in average *ScalableCA* builds 3-wise CA of 38.9% smaller size than current state-of-the-art algorithms, *i.e., SamplingCA* [61] and *CAmpactor* [97], indicating the superiority of *ScalableCA*; Also, *ScalableCA* runs one to two orders of magnitude faster than *SamplingCA* and *CAmpactor*, showing the high efficiency. In addition, our evaluation confirms the effectiveness of each novel algorithmic technique introduced by *ScalableCA*. Our experiments clearly show that *ScalableCA* effectively mitigates the scalability challenge for 3-wise CIT.

The main contributions of this work are summarized as follows.

- Through an empirical study on various real-world, highly configurable systems, we observe that 2-wise CA covers the majority of valid 3-wise tuples, and reveal that it is promising to build the 3-wise CA based on a 2-wise CA.
- Based on this key observation, we design *ScalableCA*, a new and scalable algorithm that is able to effectively alleviate the scalability challenge for 3-wise CIT.
- We propose three novel techniques, *i.e.,* fast invalidity detection, uncovering-guided sampling, and remainder-aware local search, to enhance the performance of *ScalableCA*.
- Extensive experiments show the superiority of *ScalableCA*, indicating that *ScalableCA* can considerably push forward the state of the art in solving the 3-wise CCAG problem.

2 Preliminaries

Here we provide necessary notations and definitions of this work.

2.1 Combinatorial Interaction Testing

System Under Test. A *system under test* (SUT), *i.e.,* a configurable system and an instance in this work, can be configured using a set of *option*s, denoted as O. Each option $o_i \in O$ has its *value domain* V_i, which indicates the set of all possible values for o_i. As discussed in Section 1, real-world configurable systems typically have a set of hard *constraint*s on options, denoted as H, which specifies the permissible combinations of option values. Hence, in this work an SUT S is expressed as a pair $S = (O, H)$.

Tuple. Given an SUT $S = (O, H)$, a *tuple* is a collection of pairs, *i.e.,* $\tau = \{(o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), \ldots, (o_{i_t}, v_{i_t})\}$, implying that option $o_{i_j} \in O$ takes value $v_{i_j} \in V_{i_j}$. A tuple of size t is called a *t-wise tuple*. In this work, *2-wise tuple* (*i.e.,* pairwise tuple) and *3-wise tuple* are critical concepts. Moreover, given a 3-wise tuple $\tau = \{(o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2}), (o_{i_3}, v_{i_3})\}$, τ has three derived 2-wise tuples, *i.e.,* $\{(o_{i_1}, v_{i_1}), (o_{i_2}, v_{i_2})\}$, $\{(o_{i_1}, v_{i_1}), (o_{i_3}, v_{i_3})\}$ and $\{(o_{i_2}, v_{i_2}), (o_{i_3}, v_{i_3})\}$.

Test Case. Given an SUT $S = (O, H)$, a *test case* (*i.e., configuration*) is a tuple that covers all options in O. A test case is a $|O|$-wise tuple, *i.e.,* $\pi = \{(o_1, v_1), (o_2, v_2), \ldots, (o_{|O|}, v_{|O|})\}$, indicating that option $o_i \in O$ is assigned value $v_i \in V_i$. A *test suite* is a set of test cases. A t-wise tuple τ is *covered* by a test case π if $\tau \subseteq \pi$; that is, all options in τ take the same values as the ones in π. Also, a t-wise tuple τ is *covered* by a test suite T if τ is covered by any test case in T. Given a t-wise tuple τ and a test case π, notation $\pi \circ \tau$ is a new test case π', satisfying the following requirements: 1) π' covers τ; 2) for each option o_i not appearing in τ, o_i's value in π and π' remains the same. Thus, $\pi \circ \tau$ stands for a new test case that overrides π by τ.

For many practical configurable systems, the values assigned to options are subject to hard constraints [73]. To guarantee the correctness of testing process, each adopted test case requires to satisfy all hard constraints. In this work, a test case π is *valid* if it satisfies all hard constraints. Also, a t-wise tuple τ is *valid* if τ is covered by at least one valid test case. Given an SUT S, its test suite T and a testing strength t, the t-wise coverage of T is the fraction between the number of T'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 standard and well-known metric, and it has been adopted by recent studies on testing highly configurable systems [4, 59, 69, 89].

Covering Array. Given an SUT $S = (O, H)$, a t-wise covering array (CA) is a test suite consisting of valid test cases, denoted as A, such that all valid t-wise tuples are covered by A.

The problem of t-wise constrained covering array generation (CCAG) is to build a t-wise CA as small-sized as possible, which is a fundamental problem in CIT [52, 61, 97]. As discussed in Section 1, adopting 3-wise CA for testing highly configurable systems detects over 95% of faults. However, solving large-scale 3-wise CCAG instances (*i.e.,* generating 3-wise CAs for highly configurable systems) still remains a challenge. Hence, it is crucial to design high-performance algorithms for large-scale 3-wise CCAG instances.

This work focuses on the binary scenario, where each option takes a Boolean value, following recent research [4, 59, 61, 97]. It is known that the general scenario (*i.e.,* non-binary scenario), where each option can take multiple possible values, can be converted into the binary scenario [4, 59, 61, 97]. The instances used in this work are all transformed from the non-binary scenario and collected from real-world, highly configurable systems, emphasizing the practical importance of studying the binary scenario for CIT [4, 59, 61, 97].

2.2 Boolean Formulae

It is well known that an SUT can be encoded as a Boolean formula [2, 5, 63, 64, 79]. As recognized by recent studies [4, 59, 61, 97], an effective way to deal with highly configurable systems is to utilize effective techniques for handling Boolean formulae. Hence, we introduce Boolean formulae and present the connection between Boolean formulae and highly configurable systems.

Given a Boolean *variable* x, either x or $\neg x$ is a *literal*, and a *clause* c is a disjunction of literals. Boolean variable serves as the fundamental component of a Boolean *formula*, usually expressed in conjunctive normal form (CNF) [76]. 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. Given a formula F in CNF, $V(F)$ represents the set of all Boolean variables in F while $C(F)$ is the set of all clauses in F.

Given a Boolean variable $x_i \in V(F)$, the *value* of x_i is either 0 or 1. An *assignment* of a formula F refers to a mapping $\alpha : V(F) \longrightarrow \{0, 1\}$, and each variable is assigned a Boolean value under α. For a clause $c_j \in C(F)$, c_j has two possible states under assignment α: if at least one literal in c_j evaluates to 1 under α, then c_j is *satisfied*; otherwise, c_j is *unsatisfied*. Given an assignment α, if α makes all clauses satisfied, then α is a *satisfying assignment*, also known as a *solution*; otherwise, α is an *unsatisfying assignment*.

Given an SUT $S = (O, H)$ and its encoded Boolean formula F, the option set O of S is related to the variable set $V(F)$ of F, and the hard constraint set H of S corresponds to the clause set $C(F)$ of F. Moreover, a (valid) test case of S is a (satisfying) assignment of F, and a t-wise tuple of S is F's literal combination of size t. For example, the 3-wise tuple of S, *i.e.,* $\{(o_1, 0), (o_6, 1), (o_8, 1)\}$, corresponds to the literal combination of F, *i.e.,* $\{\neg x_1, x_6, x_8\}$.

Given an SUT S and its encoded Boolean formula F, the t-wise CCAG problem in CIT is equivalent to the problem of building a set of F's satisfying assignments, ensuring that all valid t-wise tuples are covered. In theory, due to the existence of hard constraints, finding a satisfying assignment for a Boolean formula is known as the influential Boolean satisfiability (SAT) problem, which is a prototypical NP-complete problem [7]. Therefore, a practical SAT solver is necessary when solving the t-wise CCAG problem. Since a recent SAT solver named *ContextSAT* [61], which is developed on the basis of *MiniSAT* [17], exhibits its effectiveness in handling those Boolean formulae modeled from SUTs [61, 97], in this work *ScalableCA* also employs *ContextSAT* to process hard constraints.

3 Challenge, Study and Solution

Here, we first discuss the scalability challenge for 3-wise CIT, and then we perform empirical study to present a potential solution.

3.1 Scalability Challenge for 3-wise CIT

As described in Section 1, existing CCAG algorithms suffer from the scalability challenge [61, 74, 89, 97]. Recently, two powerful CCAG algorithms, *i.e., SamplingCA* [61] and *CAmpactor* [97], are proposed to mitigate the scalability challenge for pairwise testing. They represent the current state of the art of the CCAG problem and can effectively generate 2-wise CAs for highly configurable systems. However, beyond pairwise testing, there still exists the severe scalability challenge for 3-wise CIT, and both of them cannot process large-scale 3-wise CCAG instances effectively and efficiently.

When dealing with the t-wise CCAG problem, both *SamplingCA* and *CAmpactor* aim to directly build the t-wise CA, so as to cover all valid t-wise tuples. However, given a highly configurable system, the number of valid 3-wise tuples is much greater than that of valid 2-wise tuples. According to recent empirical studies on testing highly configurable systems [4, 59, 61, 97], such types of systems usually expose thousands of options. As an example, for a highly configurable system with one thousand options, where each option has 2 possible values, the number of all possible 2-wise tuples is $\binom{1000}{2} \times 2^2$, which is smaller than 2 million, while the number of all possible 3-wise tuples is $\binom{1000}{3} \times 2^3$, a huge value that is greater than 1.3 billion. For a highly configurable system, the number of 3-wise tuples is several orders of magnitude greater than that of 2-wise tuples. Also, with the growth of the number of options, the gap between the numbers of 2-wise tuples and 3-wise tuples becomes more significant. The existence of the huge number of 3-wise tuples severely degrades both effectiveness and efficiency of *SamplingCA* and *CAmpactor*, which explains the reason why *SamplingCA* and *CAmpactor* still suffer from the scalability challenge for 3-wise CIT.

Also, our experiments (Section 6) on various real-world, highly configurable systems show that *SamplingCA* and *CAmpactor* consume much running time (*e.g.,* a few days) to build 3-wise CAs of relatively large sizes. Since using a large-sized test suite would reduce the testing effectiveness in practice [70], it is critical to propose a practical solution to the scalability challenge for 3-wise CIT.

Table 1: Results of the 2-wise CAs generated by *SamplingCA* and *CAmpactor* on all large-scale instances.

	SamplingCA's 2-wise CA	*CAmpactor*'s 2-wise CA
#Total	2,180,314,493.8	2,180,314,493.8
#Cov.	2,165,423,277.7	2,120,251,170.6
#Uncov.	14,891,216.1	60,063,323.2
3-wise coverage	99.3%	97.2%

3.2 Empirical Study

Setup of Empirical Study. We conduct an empirical study to investigate whether a 2-wise CA covers the majority of valid 3-wise tuples. In this empirical study, we adopt a standard and well-known metric called 3-wise coverage (*i.e.,* t-wise coverage with $t = 3$, as described in Section 2.1) [59, 89], to assess the number of valid 3-wise tuples covered by a given test suite [59]. If a test suite achieves higher 3-wise coverage, then it covers more 3-wise tuples. To this end, the target of the empirical study is equivalent to analyzing whether a 2-wise CA could obtain high 3-wise coverage in practice.

For the empirical study, we adopt a benchmarking set of 122 large-scale, public instances, and these instances have been widely evaluated in recent studies on testing highly configurable systems [4, 59, 61, 97]. Moreover, each instance is collected from a real-world, highly configurable system, and encoded as a Boolean formula. For our adopted instances, the number of options varies from 94 to 1,850, and the number of hard constraints ranges from 190 to 62,183. To help readers better reproduce our empirical results, all instances used in this work and their related information (*i.e.,* the numbers of options and hard constraints) are public available in our repository.[1]

As shown by experiments in the literature [61, 97], current state-of-the-art algorithms (*i.e., SamplingCA* [61] and *CAmpactor* [97]) can effectively build 2-wise CAs for SUTs. Therefore, in our empirical study, we employ *SamplingCA* and *CAmpactor* for generating 2-wise CAs. Since both of them are randomized algorithms [61, 97], each algorithm is performed 10 independent runs per instance.

Results of Empirical Study. For each algorithm's built 2-wise CAs over all instances, we report the average number of covered 3-wise tuples ('#Cov.'), the average number of uncovered, valid 3-wise tuples ('#Uncov.'), and the average 3-wise coverage; also, we list the average number of all valid 3-wise tuples, denoted as '#Total'. The results are summarized in Table 1. We observe that the 2-wise CAs generated by *SamplingCA* and *CAmpactor* achieve the average 3-wise coverage of 99.3% and 97.2%, respectively. Our results confirm that 2-wise CA can cover the majority of valid 3-wise tuples in extensive real-world scenarios, indicating the feasibility of the following solution to the scalability challenge for 3-wise CIT.

3.3 Potential Solution

To relieve the scalability challenge for 3-wise CIT, a potential solution realized by our *ScalableCA* algorithm is to construct the 3-wise CA based on a 2-wise CA. The effectiveness of this solution depends on our empirical observation that a 2-wise CA covers the majority of valid 3-wise tuples. Since the 3-wise CCAG problem aims to cover all valid 3-wise tuples, compared with building 3-wise CA from the scratch, our solution (*i.e.,* generating 3-wise CA based on 2-wise CA) could greatly decrease the number of 3-wise tuples that need to be covered, which significantly reduces the problem space.

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

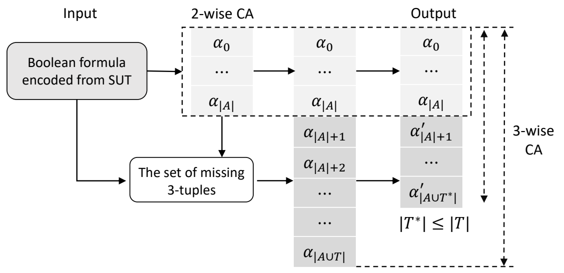

Figure 1: *ScalableCA*'s entire process for building 3-wise CA.

4 Our Proposed *ScalableCA* Algorithm

In this section, we propose *ScalableCA*, a novel and scalable algorithm for effectively tackling the 3-wise CCAG problem.

4.1 Overall Design of *ScalableCA*

Based on the observation in Section 3.2, we develop a novel and scalable algorithm called *ScalableCA* for solving the 3-wise CCAG problem. The main ideas behind *ScalableCA* are as follows: 1) an existing algorithm is invoked to build a 2-wise CA A to cover the majority of valid 3-wise tuples; 2) another test suite T is constructed to cover all the remaining, valid 3-wise tuples. In this manner, each valid 3-wise tuple is ensured to be covered by $A \cup T$, indicating that $A \cup T$ is a 3-wise CA, so *ScalableCA* reports $A \cup T$ as its final output. The overall design of *ScalableCA* is outlined in Algorithm 1. The input of *ScalableCA* is a given Boolean formula F that is encoded from an SUT, and the output of *ScalableCA* is a 3-wise CA of F.

To make readers better understand *ScalableCA*, Figure 1 illustrates its entire process for building 3-wise CA. According to Algorithm 1 and Figure 1, *ScalableCA* consists of three key stages, *i.e.,* initialization stage, sampling stage and optimization stage. In the initialization stage, *ScalableCA* generates a 2-wise CA A using an existing algorithm, and it obtains the remaining set U of all valid 3-wise tuples not covered by A (Lines 1–2 in Algorithm 1). In the sampling stage, *ScalableCA* builds another test suite T to cover all valid 3-wise tuples in U (Lines 3–14 in Algorithm 1). In the optimization stage, *ScalableCA* reduces T's size via removing and altering test cases in T, while ensuring all 3-wise tuples in U to remain covered (Line 15 in Algorithm 1).

4.2 Initialization Stage

Since a feasible solution to the scalability challenge for 3-wise CIT is to construct the 3-wise CA by extending a 2-wise CA, in the initialization stage *ScalableCA* is designed to invoke an existing algorithm to build a 2-wise CA A. As discussed before, in the literature there are two state-of-the-art algorithms, *i.e., SamplingCA* and *CAmpactor*, both of which can effectively generate a 2-wise CA for a given highly configurable system. According to Table 1, on average the 2-wise CA generated by *SamplingCA* covers more valid 3-wise tuples than the one built by *CAmpactor*; that is, adopting

Algorithm 1: Overall Design of the *ScalableCA* Algorithm

```
   Input: F: Boolean formula in CNF;
   Output: A*: 3-wise covering array (CA) of F;
1  $A \leftarrow$ the 2-wise CA built by SamplingCA;
2  $U \leftarrow$ the remaining set of all valid 3-wise not covered by $A$ via FID;
3  $T \leftarrow \varnothing$;
4  $U' \leftarrow U$;
5  while True do
6      $ugsprob \leftarrow UpdateUGSProb(F, U')$;
7      $Q \leftarrow$ a set of $\delta$ valid test cases sampled according to ugsprob;
8      $\alpha^* \leftarrow$ the test case with the largest contribution from $Q$;
9      if $contribution(\alpha^*, T) \leq 0$ then break;
10     $T \leftarrow T \cup \{\alpha^*\}$;
11     Remove all 3-wise tuples that are covered by $\alpha^*$ from $U'$;
12 foreach 3-wise tuple $\tau$ in $U'$ do
13     Generate a valid test case $\alpha$ that covers $\tau$;
14     $T \leftarrow T \cup \{\alpha\}$;
15 $T^* \leftarrow RALS(U, T)$;
16 return $A^* \leftarrow A \cup T^*$;
```

SamplingCA for generating 2-wise CA would result in fewer uncovered, valid 3-wise tuples, which could further reduce the problem space for both sampling and optimization stages. Thus, *ScalableCA* calls *SamplingCA* to generate the 2-wise CA A. The initialization stage is outlined in Lines 1–2 in Algorithm 1.

Besides building 2-wise CA, another task in this stage is to construct the remaining set U of all valid 3-wise tuples not covered by A. To obtain U, a natural approach is to conduct an enumeration process, where each possible 3-wise tuple τ is examined whether τ is valid and uncovered. In fact, justifying whether an uncovered 3-wise tuple is valid requires invoking a SAT solver one time. Since the SAT problem is NP-complete and is computationally challenging, even calling an effective SAT solver would cost a certain amount of running time [61]. Due to the existence of hard constraints, it is recognized that in practice there exist many invalid 3-wise tuples [59], so the natural approach would be quite time-consuming.

Fast Invalidity Detection Technique. To obtain U in an efficient manner, it is advisable to decrease the number of SAT solver calls. Here we propose a novel technique called fast invalidity detection (*FID*), which is designed based on the following property.

Property 1. *Given a 3-wise tuple τ, if any of τ's three derived 2-wise tuples is invalid, then τ is invalid.*

Proof. Assuming τ's one derived 2-wise tuple ω is invalid, it means that no valid test case covering ω exists. Therefore, there is no valid test case that covers τ, so τ is an invalid 3-wise tuple. □

When justifying the validity status of a 3-wise tuple τ, according to Property 1, our *FID* technique first checks whether τ's all three derived 2-wise tuples are valid; if this is the case (*i.e.,* each derived 2-wise tuple of τ is valid), then *FID* calls a SAT solver named *ContextSAT* [61] to verify the validity status of τ; otherwise, τ can be directly decided as an invalid 3-wise tuple without calling SAT solver. Hence, compared to the natural approach, *FID* could reduce the number of SAT solver calls, which could improve the efficiency for achieving U. Further, A is a 2-wise CA, so checking the validity status of a given 2-wise tuple ω is equivalent to judging whether A covers τ. As recognized by recent studies [61, 97], the process of checking whether A covers τ is efficient, which ensures that activating *FID* to construct U is of high efficiency. In fact, the efficiency of our *FID* technique will be analyzed in Section 6.3.

4.3 Sampling Stage

The sampling stage is to construct a test suite T covering all 3-wise tuples in U, such that $T \cup A$ becomes a 3-wise CA. Reducing the number of test cases is critical, since a large-sized test suite would incur inefficient testing in practice. The ultimate target of this stage is to build a test suite of small size while covering all 3-wise tuples in U. The sampling stage is outlined in Lines 3–14 in Algorithm 1.

Following the effective two-procedure design demonstrated in 2-wise CA generation [61], the sampling stage comprises two procedures, *i.e.,* iterative procedure and addition procedure. In the iterative procedure, *ScalableCA* iteratively constructs a test suite T, which is initialized as an empty set, to cover as many 3-wise tuples in U as possible. Then, in the addition procedure *ScalableCA* adds a certain number of valid test cases into T, to ensure that T covers all 3-wise tuples in U. As described above, the effectiveness of iterative procedure plays a crucial role in minimizing the final size of T, since the size of test suite produced by iterative procedure directly impacts T's size. Moreover, if the iterative procedure's built test suite covers more 3-wise tuples in U, then in the addition procedure *ScalableCA* needs fewer test cases to be inserted into T. Thus, for the iterative procedure, it is critical to generate a test suite of small size while covering more 3-wise tuples in U. The iterative procedure is presented in Lines 3–11 in Algorithm 1, while the addition procedure is shown in Lines 12–14 in Algorithm 1. This subsection describes the technical details of the iterative procedure.

In order to enhance the effectiveness of iterative procedure, in each iteration it is advisable to generate a valid test case α^*, which maximizes the benefit, and adds α^* into T. Thus, we need to address a core problem, *i.e.,* how to effectively quantify the benefit of a valid test case. As discussed above, one primary target of the iterative procedure is to empower test suite T to cover 3-wise tuples in U as many as possible, so it is desirable to design an evaluation metric that focuses on computing the increment in the number of covered 3-wise tuples that belong to U. Based on this design, we propose an effective evaluation metric called *contribution* to precisely assess the unique contribution of a given valid test case α over T with regard to U. Given a valid test case α, a test suite T, and a remaining set U, the *contribution* of α is defined as the increment in the number of T's covered 3-wise tuples that belong to U if α is added into T.

We discuss how to build a valid test case with large *contribution* in each iteration. Inspired by recent works [59, 61], *ScalableCA* uses a greedy mechanism for choosing the test case. In each iteration *ScalableCA* first constructs a candidate set Q containing δ valid test cases. From Q *ScalableCA* greedily selects the one α^* with the largest *contribution* and adds α^* into T. Here δ is an integer-valued hyper-parameter of *ScalableCA*, and the impact of δ's setting on *ScalableCA*'s performance will be studied in Section 6.4.

Also, we need to discuss the termination criterion of the iterative procedure. As the iterative procedure continues, the number of uncovered 3-wise tuples in U would be reduced, leading to smaller

Algorithm 2: The *UpdateUGSProb* Mechanism

```
Input: F: Boolean formula in CNF;
       U': current set of uncovered, valid 3-wise tuples;
Output: ugsprob: uncovering-guided sampling probabilities;
1 foreach $x_i \in V(F)$ do
2     $\lambda_i \leftarrow$ the number of $x_i$-negative 3-wise tuples in $U'$;
3     $\xi_i \leftarrow$ the number of $x_i$-positive 3-wise tuples in $U'$;
4     if $\lambda_i + \xi_i > 0$ then $ugsprob(x_i) \leftarrow \xi_i/(\lambda_i + \xi_i)$;
5     else $ugsprob(x_i) \leftarrow 0.5$;
6 return ugsprob;
```

contribution values for selected test cases in subsequent iterations. Once the *contribution* of the picked test case α^* is 0, the iterative procedure terminates. This termination criterion ensures only test cases with positive *contribution* can be inserted into T, thus preventing unnecessary expansion of T's size.

Uncovering-Guided Sampling Method. According to the description of greedy mechanism, its effectiveness is obviously impacted by the quality of candidate set Q. Hence, the generation of high-quality candidate set is a crucial problem that requires to be addressed. To generate high-quality candidate set, state-of-the-art algorithms adopt a context-aware sampling (*CAS*) method, which aims to sample a set of valid test cases that are dissimilar to those test cases already in T [59, 61]. As discussed before, it is desirable to construct Q as a set including multiple valid test cases with large *contribution*. However, existing *CAS* method does not explicitly take the remaining set U of uncovered 3-wise tuples into consideration during its process of sampling test cases, so *CAS* would not be sufficiently capable of sampling test cases with large *contribution*, which imminently calls for effective sampling methods.

To tackle this serious issue, we propose a novel uncovering-guided sampling (*UGS*) method, which focuses on sampling a collection of test cases covering more 3-wise tuples in U. Before introducing the technical details of our *UGS* method, we first present the notion of uncovering-guided sampling probability. Given a Boolean formula F, the uncovering-guided sampling probability of a variable $x_i \in V(F)$, denoted as $ugsprob(x_i)$, represents the probability that the value of x_i is sampled as 1. That is, the probability that the value of x_i is sampled as 0 is $1 - ugsprob(x_i)$.

Since *UGS* aims to sample multiple test cases that cover more 3-wise tuples in U, our *UGS* method is designed to concentrate on such currently uncovered 3-wise tuples in U. As aforementioned, in each iteration, a selected test case α^* would be added into T, making U contain fewer uncovered 3-wise tuples, so it is necessary to maintain another collection U' that consists of all 3-wise tuples that belong to U and are currently not covered by T. To this end, U' is initialized and updated as follows: 1) U' is initialized as a duplicate copy of U; 2) at the end of each iteration, the 3-wise tuples covered by α^* are removed from U'. In fact, once *ScalableCA* achieves a test suite T that makes U' become empty (*i.e.,* all 3-wise tuples in U become covered by T), $A \cup T$ would be a 3-wise CA.

Since U' plays a key role in our *UGS* method, it is desirable to make uncovering-guided sampling probabilities reflect the status of U'. Recent studies show that adaptively updating sampling probability could strengthen the effectiveness of sampling methods [59, 61]. As a result, in each iteration *UGS* dynamically updates each variable's uncovering-guided sampling probability based on the status of U'. Given a variable x_i and a tuple τ where x_i appears, if the value of x_i under τ is 0, τ is a x_i*-negative* tuple; otherwise, τ is a x_i*-positive* tuple. For each variable x_i, notation λ_i denotes the number of x_i-negative 3-wise tuples in U', and notation ξ_i represents the number of x_i-positive 3-wise tuples in U'. To sample a test case covering more 3-wise tuples in U', for each variable x_i, $ugsprob(x_i)$ should be larger if $\lambda_i < \xi_i$; otherwise, $ugsprob(x_i)$ should be smaller. Based on this discussion, in this work $ugsprob(x_i)$ is computed as the ratio between ξ_i and $\lambda_i + \xi_i$. The mechanism of updating *ugsprob*, *i.e., UpdateUGSProb*, is presented in Algorithm 2.

In each iteration, our *UGS* method updates *ugsprob* for each variable and samples multiple test cases accordingly. However, these sampled test cases are not guaranteed to be valid due to hard constraints. To ensure validity, particularly for test cases generated in the addition procedure (Line 13 in Algorithm 1), it is widely recognized that using a SAT solver is an effective method [59, 61, 97]. Hence, as described in Section 2.2, *ScalableCA* employs an effective SAT solver named *ContextSAT* [61] to generate valid test cases and alter an invalid test case into a valid one with minimal modification.

4.4 Optimization Stage

Once the sampling stage terminates, the union of the initialization stage's built 2-wise CA A and the sampling stage's constructed test suite T, *i.e.,* $A \cup T$, becomes a 3-wise CA. The optimization stage (Line 15 in Algorithm 1) is to reduce the size of generated 3-wise CA, while preserving all valid 3-wise tuples to be covered.

Local search algorithms, known for their effectiveness in reducing covering array size, achieve state-of-the-art performance [42–44, 52, 97]. Also, local search has shown great success in solving a variety of combinatorial optimization problems [11, 22, 45–51, 54–60, 77]. Notably, *CAmpactor* stands out as the current best algorithm in this domain, particularly excelling in compacting 2-wise CA [97]. Given this, a natural solution is to directly employ *CAmpactor* to compact the whole 3-wise CA. However, *CAmpactor*'s design necessitates operations on all valid 3-wise tuples. As discussed in Section 3.1, the scalability challenge for 3-wise CIT results in a vast number of valid 3-wise tuples for highly configurable systems, incurring both ineffectiveness and inefficiency of *CAmpactor*.

Remainder-Aware Local Search Approach. To address this severe problem, we design a new and effective remainder-aware local search (*RALS*) approach. Rather than *CAmpactor* that aims to compact $A \cup T$ (*i.e.,* the entire 3-wise CA), *RALS* minimizes T's size. Compared to operating on all valid 3-wise tuples of the given SUT, *RALS* only concerns the valid 3-wise tuples in the remaining set U (we note that U is different from U', which is only used in the sampling stage). *RALS* takes test suite T and remaining set U as its inputs, and it optimizes T's size while making all 3-wise tuples in U be covered. Finally, *RALS* returns an optimized test suite T^* of smaller size. Through this way, $A \cup T^*$ is guaranteed to be a 3-wise CA and is the final output of the entire *ScalableCA* algorithm.

Our *RALS* approach is presented in Algorithm 3, where *RALS* repetitively conducts search steps to modify T, in order to minimize T's size. In each search step, *RALS* first checks whether T covers all 3-wise tuples in U. If so, T^* is updated accordingly, and a random

Algorithm 3: The *RALS* Approach

```
Input: U: remaining set obtained in the initialization stage;
       T: test suite output by the sampling stage;
Output: T*: optimized test suite;
1  $T^* \leftarrow T$;
2  while $T^*$ has been updated during the last $L$ search steps do
3      if $T$ covers all 3-wise tuples in $U$ then
4          $T^* \leftarrow T$;
5          Remove a random test case from $T$;
6          continue;
7      $\tau \leftarrow$ an uncovered 3-wise tuple randomly chosen from $U$;
8      $\Theta \leftarrow \{(\beta, \beta \circ \tau) \mid \beta \in T$ is valid, and $\beta \circ \tau$ is valid$\}$;
9      if $\Theta$ is not empty then
10         $\theta^* \leftarrow$ the operation with the largest score from $\Theta$;
11         Perform operation $\theta^*$ on $T$;
12     else
13         $\beta \leftarrow$ a test case randomly picked from $T$;
14         $\beta' \leftarrow$ a valid test case covering $\tau$ found by ContextSAT;
15         Perform operation $(\beta, \beta')$ on $T$;
16 return $T^*$;
```

test case is removed from T for decreasing T's size by 1 (Lines 4–6 in Algorithm 3); otherwise, an operation is performed on T, in order to make T cover more 3-wise tuples in U (Lines 7–15 in Algorithm 3). Actually, the basic skeleton of *RALS* is to solve a series of decision problems: given a specific size μ, the decision problem is to find a test suite of size μ such that all 3-wise tuples in U are covered. Once *RALS* solves the decision problem of size μ, *RALS* continues to solve the decision problem of size $\mu - 1$. Through this way, *RALS* can reduce the size of T, and T^* represents the smallest-sized test suite that covers all 3-wise tuples in U during the search process. Our *RALS* algorithm would terminate once T^* has not been updated in the last L search steps, where L is an integer-valued hyper-parameter. Adjusting L could balance the effectiveness and efficiency of *RALS*, and its effect will be analyzed in Section 6.4.

As analyzed above, an important goal of *RALS* is to decide the operation to be performed in each search step. An operation is defined as a pair $\theta = (\alpha, \beta)$, where $\alpha \in T$ and $\beta \notin T$ are both valid test cases, and performing operation θ means replacing α with β in T. Since the decision problem aims to cover all 3-wise tuples in U, it is advisable to perform the operation that can cover more 3-wise tuples in U. Hence, we propose an effective metric called *score* to assess the benefit of an operation θ if θ is performed on T. Given an operation θ, the *score* of operation θ, denoted as $score(\theta)$, is the increment in the number of covered 3-wise tuples in U if θ is performed on T. Specifically, $score(\theta)$ is computed as the number of uncovered 3-wise tuples in U becoming covered, minus the number of covered 3-wise tuples in U becoming uncovered, if θ is performed on T. Thus, it is beneficial to perform operations with large *score*.

In each search step, for performing an operation with large *score*, *RALS* works as follows. First, a 3-wise tuple τ, which is not covered by T, is randomly chosen from U; then, *RALS* constructs a set of candidate operations Θ that aim to make τ become covered, *i.e.,* $\Theta = \{(\beta, \beta \circ \tau) \mid \beta \in T$ is valid, and $\beta \circ \tau$ is valid$\}$, where operator $\circ$ has been defined in Section 2.1, and $\beta \circ \tau$ represents a new test case that overrides β by τ. If candidate operation set Θ is not empty, *RALS* selects and performs the best operation θ^* from Θ (*i.e.,* the one with the largest *score*). Otherwise, *RALS* is considered to encounter the local optimum situation, since no candidate operation can be performed. It is recognized that employing randomized strategies could help local search algorithms better handle the local optimum situation [25, 40, 46, 47, 66, 97]. Hence, *RALS* applies the following random strategy: *RALS* randomly selects a test case β from T, and then it invokes a SAT solver named *ContextSAT* [61] to achieve a valid test case β' that covers τ; finally, *RALS* replaces β with β' in T. To this end, *RALS* is capable of escaping from local optimum.

Discussion on How *ScalableCA* Handles Constraints. As described in Section 2.2, an SUT can be encoded as a Boolean formula [2, 5, 63, 64, 79], and the constraints of SUT correspond to the clauses of Boolean formula; hence, constraints are incorporated into Boolean formula. Particularly, *SamplingCA* invokes a SAT solver named *ContextSAT* [61] to handle the encoded Boolean formula, for effectively generating valid test cases (in Sections 4.3 and 4.4) and examining tuples' validity status (in Section 4.2). Through this way, *ScalableCA* can effectively handle constraints.

5 Experimental Design

In this section, we present the experimental design of this work.

5.1 Public Instances and Competitors

We use a set of 122 large-scale, public instances, all of which are collected from practical applications. The instances adopted in our experiments are the same as the ones used in our empirical study (Section 3.2), where the information of these instances is introduced.

As discussed in Section 3.1, both *SamplingCA* [61] and *CAmpactor* [97] represent the state of the art in generating covering array. Thus, we compare *ScalableCA* against *SamplingCA* and *CAmpactor*.

SamplingCA [61] is a recently-proposed algorithm that has achieved the state-of-the-art performance in building covering arrays. The experiments in the literature [61] show that *SamplingCA* greatly outperforms various algorithms (including *AutoCCAG* [52], *FastCA* [42], *TCA* [44], *CASA* [18, 19], *HHSA* [27] and *ACTS* [94]) in generating 2-wise CA. In this work, we evaluated the latest version of *SamplingCA*, whose implementation is available online.[2]

CAmpactor [97] is the current best algorithm for compacting a given covering array. As reported in the literature [97], when constructing 2-wise CA, *CAmpactor* greatly outperforms existing algorithms, including *SamplingCA*, *AutoCCAG*, *FastCA*, *TCA*, *CASA*, *HHSA*, *ACTS* and *CTLog* [1]. In this work, *CAmpactor* is tested using its latest version, whose source code is available online.[3]

Besides *SamplingCA* and *CAmpactor*, we also compare *ScalableCA* against 7 well-known algorithms from various categories of CCAG algorithms, *i.e., Calot* [92] from the category of constraint-encoding algorithms, *AETG* [12] from the category of greedy algorithms, *ACTS* [94] and *JCunit* [82, 83] from the category of incremental generation algorithms, as well as *TCA* [44], *FastCA* [42, 43] and *AutoCCAG* [52] from the category of meta-heuristic algorithms. However, our evaluations present that, due to the severe scalability challenge, those 7 algorithms fail to generate 3-wise CAs for the

[2]https://github.com/chuanluocs/SamplingCA/tree/general
[3]https://github.com/chuanluocs/CAmpactor/tree/general

majority of large-scale instances. To save space, we do not present the results of these 7 algorithms in this paper. The results of these 7 algorithms are publicly available at our repository.[1]

5.2 Research Questions

Since this work is devoted to alleviating the severe scalability challenge for 3-wise CIT, our experiments target to minimize the size of constructed 3-wise CA and meanwhile reduce the running time on large-scale instances. Our experiments aim to answer the following research questions (RQs).

RQ1: Can *ScalableCA* construct 3-wise CA of smaller size than its state-of-the-art competitors on large-scale instances?

In this RQ, we evaluate the sizes of the 3-wise CAs generated by *ScalableCA*, *SamplingCA* and *CAmpactor*, on large-scale instances.

RQ2: Does *ScalableCA* require less running time to build 3-wise CA compared to its state-of-the-art competitors on large-scale instances?

In this RQ, we empirically compare the running time of *ScalableCA* against that of *SamplingCA* and *CAmpactor* on large-scale instances.

RQ3: Does each core technique proposed in this work contribute to the performance improvement of *ScalableCA*?

In this RQ, we empirically analyze the contribution made by each core technique to *ScalableCA*'s performance improvement.

RQ4: How does the setting of each hyper-parameter affect the practical performance of *ScalableCA*?

In this RQ, we empirically investigate how the settings of hyper-parameters (*i.e.,* δ and L) impact *ScalableCA*'s performance.

5.3 Experimental Setup

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

ScalableCA and its competitors are all randomized algorithms, so each competing algorithm is performed 10 independent runs per instance. As discussed in Section 3.1, due to the scalability challenge for 3-wise CIT, all *ScalableCA*'s competitors (*i.e., SamplingCA*, *CAmpactor* and other 7 competitors) are ineffective in building 3-wise CAs for highly configurable systems. According to our preliminary experiments, *SamplingCA* and *CAmpactor* could construct 3-wise CAs for all adopted instances within 2 CPU days. Hence, to make them successfully generate 3-wise CAs, in our experiments the cutoff time for each algorithm run is set to 172,800 CPU seconds (*i.e.,* 2 CPU days). Besides *SamplingCA* and *CAmpactor*, our evaluations show that the other 7 competitors cannot generate 3-wise CAs for the majority of large-scale instances within the cutoff time of 2 CPU days, and their detailed experimental results on all large-scale instances are publicly available at our repository.[1] For *ScalableCA*, we set its hyper-parameters δ and L to 100 and 500, respectively. The effects of δ and L will be analyzed in Section 6.4. For *SamplingCA* and *CAmpactor*, we adopt the hyper-parameter settings suggested by their respective authors [61, 97].

Following the recent work [97], for each competing algorithm on solving each instance, we present the minimum size of the constructed 3-wise CAs among 10 independent runs, denoted as 'min.', the average size of the constructed 3-wise CAs over 10 runs, denoted as 'avg.', and the average running time over 10 runs, denoted

Table 2: Average size and average time of *ScalableCA*, *SamplingCA* and *CAmpactor* over all large-scale instances

	ScalableCA	*SamplingCA*	*CAmpactor*
avg. size	**526.5**	903.9	862.1
avg. time (sec)	1,778.3	67,198.6	167,433.9

Table 3: Comparative results of *ScalableCA*, *SamplingCA* and *CAmpactor* on 10 representative, large-scale instances.

Instance	*ScalableCA*	*SamplingCA*	*CAmpactor*
	min. (avg.) time (sec)	min. (avg.) time (sec)	min. (avg.) time (sec)
`dreamcast`	**588 (608.0)**	990 (1006.0)	953 (971.4)
	1,717.5	73,565.4	171,407.3
`ecos-icse11`	**517 (531.1)**	896 (926.8)	856 (886.4)
	1,771.7	67,822.0	171,864.0
`freebsd-icse11`	**589 (616.0)**	1,099 (1,109.4)	1,085 (1,097.2)
	1,106.7	115,131.6	171,061.0
`integrator_arm9`	**700 (728.2)**	1,134 (1,151.5)	1,107 (1,127.0)
	1,710.3	88,026.4	171,482.7
`linux`	**567 (604.6)**	1001 (1010.8)	963 (974.0)
	1700.7	72198.3	171298.8
`mpc50`	**474 (499.7)**	828 (836.5)	772 (782.4)
	1,566.1	56,886.3	171,534.7
`ocelot`	**521 (541.2)**	920 (930.4)	881 (893.2)
	1,939.9	71,603.6	171,579.5
`pc_i82544`	**526 (556.3)**	964 (980.7)	929 (946.0)
	2,178.4	74,234.9	171,460.5
`refidt334`	**545 (575.0)**	980 (998.9)	945 (965.7)
	2,162.7	75,843.8	171,864.3
`XSEngine`	**526 (544.6)**	885 (917.0)	846 (878.7)
	1,683.5	71,305.7	171,299.4

by 'time'. Further, to evaluate the overall performance, for each algorithm, we report the average size (denoted by 'avg. size') and average running time (denoted by 'avg. time') to build 3-wise CA over the set of large-scale instances. All running times are measured in CPU second. For each instance or the set of large-scale instances, if a competing algorithm builds the smallest-sized 3-wise CA, then its results of 'min.', 'avg.' and 'avg. size' are indicated in **boldface**.

Moreover, for each large-scale instance or the set of large-scale instances, we conduct the Wilcoxon signed-rank test [16] to determine the statistical significance of pairwise comparisons between *ScalableCA* and each of its competitors, and we compute the Vargha-Delaney effect sizes [84] for all pairwise comparisons. Particularly, we consider the following criteria: 1) all p-values of the Wilcoxon signed-rank tests at a 95% confidence level are less than 0.05, and 2) the Vargha-Delaney effect sizes for all pairwise comparisons are greater than 0.71, which is known to imply large effect sizes [52, 59, 61, 78, 84, 97]. If both criteria are met, we conclude that the performance improvement of *ScalableCA* over its competitors is both statistically significant and meaningful, and *ScalableCA*'s results of 'min.', 'avg.' and 'avg. size' are highlighted via underline.

6 Experimental Results

In this section, we report and discuss the experimental results.

6.1 RQ1: Comparison on Size of 3-wise CA

We compare *ScalableCA* against *SamplingCA*, *CAmpactor* and other 7 competitors on the set of 122 large-scale, public instances. Due to page limit, we do not report the full results on all large-scale instances in this paper. The full experimental results of *ScalableCA* and all its competitors on all 122 large-scale instances are publicly available in our repository.[1] Nevertheless, to present the overall performance of *ScalableCA* and its two competitors (*i.e., SamplingCA* and *CAmpactor*), we summarize the average size and the average running time of *ScalableCA*, *SamplingCA* and *CAmpactor* on all instances in Table 2. Also, to study the performance of on a per-instance basis, the results of *ScalableCA*, *SamplingCA* and *CAmpactor* on 10 selected instances are reported in Table 3; these 10 selected instances are recognized as representative ones according to recent studies on testing highly configurable systems [59, 97].

From Tables 2 and 3, our *ScalableCA* algorithm stands out as the best algorithm, and it constructs 3-wise CA of much smaller size, compared to its state-of-the-art competitors, which presents *ScalableCA*'s effectiveness. In particular, over the set of large-scale instances, the average size of *ScalableCA*'s generated 3-wise CAs is 526.5, while this number for *SamplingCA* and *CAmpactor* is 903.9 and 862.1, respectively. Hence, *ScalableCA* generates 3-wise CA of 38.9% smaller size than current state-of-the-art algorithms, indicating the superiority of *ScalableCA* over its competitors. Our results in Tables 2 and 3 demonstrate that *ScalableCA* considerably pushes forward the state of the art in solving the 3-wise CCAG problem.

In addition, we empirically study whether *ScalableCA*'s built 3-wise CA could obtain high t-wise coverage with $4 \le t \le 6$. To save space, the related results are publicly available at our repository.[1]

6.2 RQ2: Comparison on Running Time

In this subsection, we analyze the efficiency of *ScalableCA*. Tables 2 and 3 also present the average running time of *ScalableCA* and its competitors. Table 2 shows that, over the set of large-scale instances, the average running time of *ScalableCA* is 1,778.3 seconds, while that of *SamplingCA* and *CAmpactor* is 67,198.6 seconds and 167,433.9 seconds, respectively. Moreover, from Table 3, *ScalableCA* requires much less running time than *SamplingCA* and *CAmpactor* on 10 selected instances. Thus, according to Tables 2 and 3, when building 3-wise CAs for highly configurable systems, *ScalableCA* runs one to two orders of magnitude faster than its state-of-the-art competitors, which presents the high efficiency of *ScalableCA*.

To this end, compared to current state-of-the-art algorithms, *ScalableCA* can construct 3-wise CA of significantly smaller size, using much less running time, which confirms that *ScalableCA* is able to effectively alleviate the scalable challenge for 3-wise CIT.

6.3 RQ3: Effectiveness of Each Core Technique

Here we empirically study the effectiveness of each core technique proposed by *ScalableCA*. As introduced in Section 4, *ScalableCA* introduces three novel, core techniques, *i.e.,* fast invalidity detection (*FID*) technique in the initialization stage (Section 4.2), uncovering-guided sampling (*UGS*) method in the sampling stage (Section 4.3), and remainder-aware local search (*RALS*) approach in the optimization stage (Section 4.4). To study the effectiveness of *FID*, based on *ScalableCA* we develop an alternative version named *Alt-1*, which directly works without *FID*. For investigating the contribution made by *UGS*, we design an alternative version of *ScalableCA* called *Alt-2*, which replaces *UGS* with the existing context-aware sampling (*CAS*) method [59, 61] (as discussed in Section 4.3). To analyze the effectiveness of *RALS*, we develop two alternative versions, *i.e., Alt-3* and *Alt-4*. *Alt-3* is *ScalableCA*'s alternative version that works without *RALS*. *Alt-4* is *ScalableCA*'s alternative version that replaces *RALS* with *CAmpactor*; that is, *Alt-4* employs *CAmpactor* to optimize the whole 3-wise CA built by *ScalableCA*'s sampling stage. Further, as discussed in Section 4.2, *ScalableCA* invokes *SamplingCA* to build 2-wise CA in the initialization stage. Actually, there is another alternative version *Alt-5* that calls *CAmpactor* for 2-wise CA generation. Here, *ScalableCA* is compared with all its five alternative versions.

Table 4: Average size and average time of *ScalableCA* and its alternative versions over all large-scale instances

.

	ScalableCA	*Alt-1*	*Alt-2*	*Alt-3*	*Alt-4*	*Alt-5*
avg. size	**526.5**	**526.5**	618.6	780.8	714.6	650.4
avg. time (sec)	1,778.3	2,867.2	9,059.1	772.3	167,101.3	22,764.7

Table 5: Average size and average time of *ScalableCA* with various settings of δ over all large-scale instances.

	$\delta = 10$	$\delta = 50$	$\delta = 100$	$\delta = 500$	$\delta = 1,000$
avg. size	553.5	533.4	526.5	513.7	**504.1**
avg. time (sec)	1,796.8	1,751.8	1,778.3	2,235.6	2,969.9

Table 6: Average size and average time of *ScalableCA* with various settings of L over all large-scale instances.

	$L = 100$	$L = 250$	$L = 500$	$L = 750$	$L = 1,000$
avg. size	606.1	551.1	526.5	515.3	**508.4**
avg. time (sec)	1,089.4	1,387.8	1,778.3	1,925.5	2,234.2

Table 4 presents the average size and the average running time of *ScalableCA* and all its alternative versions. From Table 4, although *ScalableCA* and *Alt-1* achieve the same average size, the average running time of *ScalableCA* and *Alt-1* is 1778.3 seconds and 2867.2 seconds, respectively; this is not surprising, since *FID* only accelerates the process of obtaining the remaining set U (as discussed in Section 4.2). Further, according to our statistical analysis, the average numbers of SAT solver calls with and without the *FID* technique are 14,901,965.7 and 398,744,187.5, respectively, and this is the reason why *FID* greatly improves the efficiency of *ScalableCA*.

Also, Table 4 shows that *ScalableCA* generates much smaller-sized 3-wise CA than *Alt-2*, *Alt-3* and *Alt-4*, indicating the effectiveness of *UGS* and *RALS*. Here we discuss the efficiency. From Table 4, it is not surprising that *ScalableCA* needs more running time than *Alt-3*, since there is no optimization stage in *Alt-3*. The pairwise comparison between *ScalableCA* and *Alt-3* shows that, on average with around 1,000 seconds more, *RALS* helps *ScalableCA* reduce the generated 3-wise CA's size by more than 250 (*i.e.,* decreasing the size by 32.6%). In practice, the size of test suite directly affects the testing budget [43, 44, 52, 70], confirming the effectiveness of *RALS*. Moreover, *ScalableCA* runs much faster than *Alt-2* and *Alt-4*, showing the high efficiency of *UGS* and *RALS*. When compared to

Table 7: Comparative results of *ScalableCA*, *SamplingCA*, *CAmpactor* and *AutoCCAG* on 5 small-scale instances.

Instance	*ScalableCA*	*SamplingCA*	*CAmpactor*	*AutoCCAG*
	min. (avg.) time (sec)	min. (avg.) time (sec)	min. (avg.) time (sec)	min. (avg.) time (sec)
apache	**134** (**134.7**) 931.63	238 (242.9) 180.44	185 (196.6) 1024.55	135 (136.6) 837.16
bugzilla	**48** (**48.0**) 45.21	82 (84.0) 2.03	**48** (**48.0**) 16.89	**48** (**48.0**) 10.77
gcc	**72** (**73.1**) 892.12	128 (130.2) 115.22	86 (90.6) 1108.51	73 (74.7) 773.83
spins	**80** (**80.0**) 268.13	118 (120.4) 0.51	**80** (81.3) 94.65	**80** (**80.0**) 1.42
spinv	**190** (**190.3**) 342.01	282 (286.1) 17.76	199 (203.6) 1179.74	191 (192.9) 947.13

Alt-5, on average *ScalableCA* takes much less running time to construct 3-wise CA of greatly smaller size, presenting the substantial advantage of adopting *SamplingCA* for building 2-wise CA.

In summary, the results in Table 4 confirm that each core technique contributes to the performance improvement of *ScalableCA*.

6.4 RQ4: Impacts of Hyper-Parameter Settings

ScalableCA has two hyper-parameters, *i.e.,* δ and L. According to Sections 4.3 and 4.4, δ determines the cardinality of candidate set in the sampling phase, while L controls the termination criterion of *ScalableCA*. Here, we analyze the impacts of the settings of δ and L.

Table 5 reports the results of *ScalableCA* with different settings of δ (*i.e.,* setting δ to 10, 50 100, 500 and $1,000$). From 5, *ScalableCA* exhibits both effectiveness and efficiency when δ is set to 100. Thus, the guideline of determining δ's value is to set $\delta = 100$.

Also, Table 6 presents the results of *ScalableCA* with different settings of L (*i.e.,* setting L to 100, 250, 500, 750 and $1,000$). As observed in Table 6, if L is set to a larger value, then *ScalableCA* can build 3-wise CA of smaller size while it requires more running time; otherwise, *ScalableCA* runs much faster while the generated 3-wise CA is of larger size. Our results indicate the flexibility of *ScalableCA*, since adjusting L can balance the effectiveness and the efficiency of *ScalableCA*. Therefore, the guideline of setting L's value is as follows. If *ScalableCA* is applied in the scenario which desires small test suite, we recommend setting L to a large value (*e.g.,* $L = 1,000$); otherwise, if rapid generation of test suite is required, we recommend setting L to a small value (*e.g.,* $L = 100$). Besides, if *ScalableCA* is adopted in a scenario where both effectiveness and efficiency are considered, we recommend setting $L = 500$.

7 Discussions

In this section, we discuss *ScalableCA*'s performance on small-scale instances and *ScalableCA*'s fault detection capability. In addition, we discuss the treats to the validity of this work.

7.1 Evaluation on Small-scale Instances

In addition to evaluating *ScalableCA* on large-scale instances, it is also interesting to analyze its effectiveness on small-scale instances. We include 25 small-scale, publicly available instances from our repository[1] for this purpose, which have also been utilized in the original papers of *AutoCCAG* [52] and *FastCA* [42], both of which are competitors of *ScalableCA* (as detailed in Section 5.1).

Compared to those 122 large-scale instances described in Section 5.1 (the average number of options is 1228.88), these 25 instances are of much smaller scale (the average number of options is 31.24). Besides, these 25 small-scale instances are of non-binary scenario, where each option can be assigned to multiple possible values. To enable *ScalableCA* to handle non-binary instances, given a non-binary instance, *ScalableCA* first converts it into a Boolean formula through the model flattening technique [23], where each value of a non-binary option in the given non-binary instance is represented by a binary variable in the converted Boolean formula, and then *ScalableCA* processes the converted Boolean formula. We note that, after converting those 25 small-scale instances into Boolean formulae, the average number of options (*i.e.,* the average number of variables) over all 25 Boolean formulae is 61.96, which is still considerably smaller than the average number of options over the set of 122 large-scale instances. Further, when handling those 25 small-scale instances, *ScalableCA* activates *AutoCCAG* [52] after the optimization stage of *ScalableCA* (Section 4.4) completes.

We evaluate *ScalableCA*, *SamplingCA*, *CAmpactor*, *AutoCCAG* and other 7 competitors on the collection of 25 small-scale instances. Due to page limit, we do not report the full experimental results in this paper. The complete and detailed experimental results are publicly available at our repository.[1] Table 7 reports the comparative results of *ScalableCA*, *SamplingCA*, *CAmpactor* and *AutoCCAG* on 5 small-scale instances out of the entire collection. According to the comparative results, *ScalableCA* builts 3-wise CAs of the smallest sizes compared to all its competitors, thus indicating that *ScalableCA* exhibits effectiveness in handling small-scale instances.

7.2 Study on Fault Detection Capability

In this subsection, we conduct an empirical evaluation to assess *ScalableCA*'s fault detection capability on real-world, highly configurable systems. In order to achieve this target, we employ 9 real-world, highly configurable systems as our subjects, all of which are originally collected and utilized by a recent empirical study [87]. Each of these subjects consists of a model file describing option information, a constraint file detailing constraint information, and a ground-truth file (*i.e.,* a file recording a list of ground-truth tuples that can trigger faults, essential for calculating the fault detection rate). We note that these 9 subjects adopted in our evaluation are publicly available at our repository.[1]

Our evaluation adopts fault detection rate ('FDR') and size of CA as assessment metrics. Given a subject and a CA T, T's fault detection rate on the subject is calculated as the ratio between the number of T's detected faults and the total number of faults that are recorded in the subject's ground-truth file. Table 8 reports the average fault detection rate and average size of 3-wise CAs built by *ScalableCA*, *SamplingCA* and *CAmpactor*, as well as 2-wise CAs built by *SamplingCA* and *CAmpactor* over 9 subjects. The detailed results on each subject are publicly available at our repository.[1] From Table 8, it is clear that 3-wise CA exhibits much stronger fault detection capability than 2-wise CA, which confirms the significance of generating 3-wise CA. Also, Table 8 shows that the 3-wise CA built by *ScalableCA* can disclose more faults than the 2-wise CAs and 3-wise CAs built by *SamplingCA* and *CAmpactor*, indicating the practical value of *ScalableCA* in real-world applications.

Table 8: Average FDR and average size of CAs by *ScalableCA*, *SamplingCA* and *CAmpactor* over 9 subjects.

	3-wise CA			2-wise CA	
	ScalableCA	*SamplingCA*	*CAmpactor*	*SamplingCA*	*CAmpactor*
avg. FDR	96.8%	94.3%	93.8%	80.0%	80.7%
avg. size	70.7	80.5	73.1	24.7	22.3

7.3 Threats to Validity

Two potential threats to the validity of this work are identified.

Generality of Instances. To make our evaluation thorough, it is critical to keep the generality of our used instances. We employ a diverse set of 122 large-scale instances and 25 small-scale instances, all of which are encoded from real-world highly configurable systems. These instances encompass a wide spectrum of options and constraints and have been extensively evaluated [4, 30, 41, 42, 52, 59, 61, 69, 74, 75], so they are general and representative. Thus, this potential threat can be mitigated.

Random Characteristic of Competing Algorithms. In our experiments, all competing algorithms are randomized, relying on a single run per instance may not yield precise evaluations. To address this issue, we conduct 10 independent runs for each algorithm per instance, following recent studies [43, 44, 52]. Further, we perform significance test and compute effect size for analyzing the comparative results, thereby reducing this potential threat.

8 Related Work

Combinatorial interaction testing (CIT) is pivotal in software testing with extensive studies (*e.g.,* [32, 68, 80, 95]). Although pairwise testing (*i.e.,* 2-wise CIT) is popular, empirical studies on extensive highly configurable systems show that 3-wise CIT reveals more faults, emphasizing its importance [33–36, 43, 52].

Constrained covering array generation (CCAG) is the core problem in CIT [44, 61, 97], and practical CCAG algorithms can be grouped into four categories, *i.e.,* constraint-encoding algorithms (*e.g.,* [1, 3, 24, 92, 96]), greedy algorithms (*e.g.,* [8–10, 12, 81, 91]), incremental generation algorithms (*e.g.,* [29, 31, 37–39, 82, 83, 85, 94]), and meta-heuristic algorithms (*e.g.,* [8, 13–15, 18, 20, 21, 27, 43, 44, 52, 62, 90]). Constraint-encoding algorithms (*e.g., Calot* [92]) translate CCAG into other optimization problems but struggle with large-scale instances [1, 3, 24, 92, 96]. Greedy algorithms (*e.g., AETG* [12]) employ the one-test-at-a-time (OTAT) strategy for constructing CAs in a greedy manner. Incremental generation algorithms (*e.g., ACTS* [94] and *JCunit* [82, 83]) adopt the in-parameter-order (IPO) technique and operate iteratively on existing CAs. Initially, they generate a CA for a small subset of all options, and then they iteratively introduce new options until a complete CA for all options is achieved. While effective for managing medium-scale instances, these two types of methods usually generate large CAs, thereby limiting their applicability in time-constrained scenarios. Meta-heuristic algorithms (*e.g., TCA* [44], *FastCA* [42, 43] and *AutoCCAG* [52]), which conduct advanced meta-heuristic search techniques, can generate small CAs, but they cost fairly long running time.

Existing CCAG algorithms suffer from the severe scalability challenge. With many options exposed by highly configurable systems, these algorithms usually need considerable running time to produce large-sized CAs, rendering testing inefficient. To mitigate the scalability challenge, two recent cutting-edge CCAG algorithms, *i.e., SamplingCA* [61] and *CAmpactor* [97], have been developed. However, both *SamplingCA* and *CAmpactor* only address the scalability challenge for pairwise testing, and the serious scalability challenge still persists for 3-wise CIT, as highly configurable systems involve immense numbers of valid 3-wise tuples. Our experimental results (Section 6) confirm that both *SamplingCA* and *CAmpactor* exhibit limitations in effectively and efficiently building 3-wise CAs for various real-world, highly configurable systems, which urgently calls for practical solutions to the 3-wise CCAG problem.

This work proposes *ScalableCA*, a novel and scalable algorithm that demonstrates effectiveness and efficiency in solving large-scale 3-wise CCAG instances. In contrast to existing CCAG algorithms, *ScalableCA* can generate small-sized 3-wise CAs efficiently for highly configurable systems, indicating that *ScalableCA* effectively alleviates the scalability challenge for 3-wise CIT. Therefore, the adoption of *ScalableCA* could show advantages in practice.

9 Conclusion

In this work, we aim to alleviate the severe scalability challenge for 3-wise CIT. Through an empirical study on various highly configurable systems, we observe that 2-wise CA covers the majority of valid 3-wise tuples. Hence, a potential solution to the scalability challenge for 3-wise CIT is to build 3-wise CA by extending 2-wise CA. Based on this potential solution, we propose a scalable algorithm dubbed *ScalableCA*. Further, *ScalableCA* proposes three novel techniques, *i.e.,* fast invalidity detection, uncovering-guided sampling, and remainder-aware local search, to enhance its performance. Our experiments on extensive large-scale instances show that, compared to existing state-of-the-art algorithms, *ScalableCA* runs one to two orders of magnitude faster to generate 3-wise CA of 38.9% smaller size in average, indicating that *ScalableCA* can effectively mitigate the scalability challenge for 3-wise CIT.

For future work, we plan to adopt *ScalableCA* to test highly configurable systems in practice. Also, we plan to use automatic tuning tools (*e.g., SMAC* [26] and *TPE* [6]) to configure *ScalableCA*.

10 Data Availability

The implementation of *ScalableCA*, all instances and results are publicly available at **https://github.com/chuanluocs/ScalableCA** and and archived at Zenodo [53].

Acknowledgments

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 Grant 62202025 and Grant 62302528, in part by the Young Elite Scientist Sponsorship Program by CAST under Grant YESS20230566, in part by the CCF-Huawei Populus Grove Fund under Grant CCF-HuaweiSY202311, in part by the Frontier Cross Fund Project of Beihang University, in part by the Fundamental Research Fund Project of Beihang University, in part by the Open Project of Xiangjiang Laboratory under Grant 22XJ03010, and in part by the Natural Science Foundation of Changsha under Grant kq2202104.

References

[1] Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. 2022. Incomplete MaxSAT approaches for combinatorial testing. *Journal of Heuristics* 28, 4 (2022), 377–431.

[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer.

[3] Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. 2010. Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers. In *Proceedings of LPAR 2010*. 112–126.

[4] Eduard Baranov, Axel Legay, and Kuldeep S. Meel. 2020. Baital: An Adaptive Weighted Sampling Approach for Improved t-wise Coverage. In *Proceedings of ESEC/FSE 2020*. 1114–1126.

[5] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of SPLC 2005*. 7–20.

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Proceedings of NIPS 2011*. 2546–2554.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2021. *Handbook of Satisfiability - Second Edition.* Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press.

[8] 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.

[9] Renée C. Bryce and Charles J. Colbourn. 2009. A Density-based Greedy Algorithm for Higher Strength Covering Arrays. *Software Testing, Verification and Reliability* 19, 1 (2009), 37–53.

[10] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. 2005. A Framework of Greedy Methods for Constructing Interaction Test Suites. In *Proceedings of ICSE 2005*. 146–155.

[11] Yi Chu, Shaowei Cai, and Chuan Luo. 2023. NuWLS: Improving Local Search for (Weighted) Partial MaxSAT by New Weighting Techniques. In *Proceedings of AAAI 2023*. 3915–3923.

[12] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. 1997. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.

[13] Myra B. Cohen, Charles J. Colbourn, and Alan C. H. Ling. 2003. Augmenting Simulated Annealing to Build Interaction Test Suites. In *Proceedings of ISSRE 2003*. 394–405.

[14] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. In *Proceedings ICSE 2003*. 38–48.

[15] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, Charles J. Colbourn, and James S. Collofello. 2003. A Variable Strength Interaction Testing of Components. In *Proceedings of COMPAC 2003*. 413–418.

[16] W. J. Conover. 1999. *Practical Nonparametric Statistics.* Conover.

[17] Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Proceedings of SAT 2003*. 502–518.

[18] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. 2009. An Improved Meta-Heuristic Search for Constrained Interaction Testing. In *Proceedings of International Symposium on Search Based Software Engineering 2009*. 13–22.

[19] 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, 1 (2011), 61–102.

[20] Syed A. Ghazi and Moataz A. Ahmed. 2003. Pair-wise Test Coverage using Genetic Algorithms. In *Proceedings of CEC 2003*. 1420–1424.

[21] Loreto Gonzalez-Hernandez, Nelson Rangel-Valdez, and Jose Torres-Jimenez. 2010. Construction of Mixed Covering Arrays of Variable Strength Using a Tabu Search Approach. In *Proceedings of COCOA 2010*. 51–64.

[22] Jiazhen Gu, Chuan Luo, Si Qin, Bo Qiao, Qingwei Lin, Hongyu Zhang, Ze Li, Yingnong Dang, Shaowei Cai, Wei Wu, Yangfan Zhou, Murali Chintalapati, and Dongmei Zhang. 2020. Efficient Incident Identification from Multi-dimensional Issue Reports via Meta-heuristic Search. In *Proceedings of ESEC/FSE 2020*. 292–303.

[23] Christopher Henard, Mike Papadakis, and Yves Le Traon. 2015. Flattening or not of the combinatorial interaction testing models?. In *Proceedings of ICST Workshops 2015*. 1–4.

[24] Brahim Hnich, Steven David Prestwich, Evgeny Selensky, and Barbara M. Smith. 2006. Constraint Models for the Covering Test Problem. *Constraints* 11, 2-3 (2006), 199–219.

[25] Holger H. Hoos and Thomas Stützle. 2004. *Stochastic Local Search: Foundations & Applications.* Elsevier / Morgan Kaufmann.

[26] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. In *Proceedings of LION 2011*. 507–523.

[27] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of ICSE 2015*. 540–550.

[28] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. 2019. Distance-based Sampling of Software Configuration Spaces. In *Proceedings of ICSE 2019*. 1084–1094.

[29] Ludwig Kampel, Bernhard Garn, and Dimitris E. Simos. 2017. Combinatorial Methods for Modelling Composed Software Systems. In *Proceedings of ICSTW 2017*. 229–238.

[30] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch between Real-world Feature Models and Product-line Research?. In *Proceedings of ESEC/FSE 2017*. 291–302.

[31] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: yet another sampling algorithm. In *Proceedings of VaMoS 2020*. 4:1–4:10.

[32] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to Combinatorial Testing.* CRC press.

[33] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421.

[34] Rick Kuhn and Raghu Kacker. 2011. Practical combinatorial (t-way) methods for detecting complex faults in regression testing. In *Proceedings of ICSM 2011*. 599.

[35] Rick Kuhn, Raghu Kacker, Yu Lei, and Justin Hunter. 2009. Combinatorial Software Testing. *Computer* 42, 8 (2009), 94–96.

[36] Rick Kuhn, Yu Lei, and Raghu Kacker. 2008. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional* 10, 3 (2008), 19–23.

[37] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2007. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of ECBS 2007*. 549–556.

[38] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148.

[39] Yu Lei and Kuo-Chung Tai. 1998. In-Parameter-Order: A Test Generation Strategy for Pairwise Testing. In *Proceedings of HASE 1998*. 254–261.

[40] Chu Min Li and Wenqi Huang. 2005. Diversification and Determinism in Local Search for Satisfiability. In *Proceedings of SAT 2005*. 158–172.

[41] 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 SPLC 2015*, Douglas C. Schmidt (Ed.). 91–100.

[42] 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 *Proceedings of ICSE 2021 (Companion Volume)*. 77–80.

[43] 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 ESEC/FSE 2019*. 212–222.

[44] 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. In *Proceedings of ASE 2015*. 494–505.

[45] Chuan Luo, Shaowei Cai, Kaile Su, and Wenxuan Huang. 2017. CCEHC: An Efficient Local Search Algorithm for Weighted Partial Maximum Satisfiability. *Artificial Intelligence* 243 (2017), 26–44.

[46] Chuan Luo, Shaowei Cai, Kaile Su, and Wei Wu. 2015. Clause States Based Configuration Checking in Local Search for Satisfiability. *IEEE Transactions on Cybernetics* 45, 5 (2015), 1014–1027.

[47] Chuan Luo, Shaowei Cai, Wei Wu, Zhong Jie, and Kaile Su. 2015. CCLS: An Efficient Local Search Algorithm for Weighted Maximum Satisfiability. *IEEE Transactions on Computers* 64, 7 (2015), 1830–1843.

[48] Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. 2013. Focused Random Walk with Configuration Checking and Break Minimum for Satisfiability. In *Proceedings of CP 2013*. 481–496.

[49] Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su. 2014. Double Configuration Checking in Stochastic Local Search for Satisfiability. In *Proceedings of AAAI 2014*. 2703–2709.

[50] Chuan Luo, Holger H. Hoos, and Shaowei Cai. 2020. PbO-CCSAT: Boosting Local Search for Satisfiability Using Programming by Optimisation. In *Proceedings of PPSN 2020*. 373–389.

[51] Chuan Luo, Holger H. Hoos, Shaowei Cai, Qingwei Lin, Hongyu Zhang, and Dongmei Zhang. 2019. Local Search with Efficient Automatic Configuration for Minimum Vertex Cover. In *Proceedings of IJCAI 2019*. 1297–1304.

[52] 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 *Proceedings of ICSE 2021*. 201–212.

[53] Chuan Luo, Shuangyu Lyu, Qiyuan Zhao, Wei Wu, Hongyu Zhang, and Chunming Hu. 2024. Artifact for ISSTA 2024 Article 'Beyond Pairwise Testing: Advancing 3-wise Combinatorial Interaction Testing for Highly Configurable Systems'. https://doi.org/10.5281/zenodo.12661302

[54] Chuan Luo, Bo Qiao, Xin Chen, Pu Zhao, Randolph Yao, Hongyu Zhang, Wei Wu, Andrew Zhou, and Qingwei Lin. 2020. Intelligent Virtual Machine Provisioning in Cloud Computing. In *Proceedings of IJCAI 2020*. 1495–1502.

[55] Chuan Luo, Bo Qiao, Wenqian Xing, Xin Chen, Pu Zhao, Chao Du, Randolph Yao, Hongyu Zhang, Wei Wu, Shaowei Cai, Bing He, Saravanakumar Rajmohan, and Qingwei Lin. 2021. Correlation-Aware Heuristic Search for Intelligent Virtual

Machine Provisioning in Cloud Systems. In *Proceedings of AAAI 2021*. 12363–12372.

[56] 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 Transactions on Software Engineering and Methodology* (2024).

[57] Chuan Luo, Kaile Su, and Shaowei Cai. 2012. Improving Local Search for Random 3-SAT Using Quantitative Configuration Checking. In *Proceedings of ECAI 2012*. 570–575.

[58] Chuan Luo, Kaile Su, and Shaowei Cai. 2014. More efficient two-mode stochastic local search for random 3-satisfiability. *Applied Intelligence* 41, 3 (2014), 665–680.

[59] 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 ESEC/FSE 2021*. 1081–1092.

[60] Chuan Luo, Wenqian Xing, Shaowei Cai, and Chunming Hu. 2024. NuSC: An Effective Local Search Algorithm for Solving the Set Covering Problem. *IEEE Transactions on Cybernetics* 54, 3 (2024), 1403–1416.

[61] 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 ESEC/FSE 2022*. 1185–1197.

[62] James D. McCaffrey. 2009. Generation of Pairwise Test Sets Using a Genetic Algorithm. In *Proceedings of COMPSAC 2009*. 626–631.

[63] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of ICSE 2016*. 643–654.

[64] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based Analysis of Feature Models is Easy. In *Proceedings of SPLC 2009*. 231–240.

[65] Hanefi Mercan, Cemal Yilmaz, and Kamer Kaya. 2019. CHiP: A Configurable Hybrid Parallel Covering Array Constructor. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1270–1291.

[66] Wil Michiels, Emile H. L. Aarts, and Jan H. M. Korst. 2007. *Theoretical aspects of local search*. Springer.

[67] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Sven Apel, and Norbert Siegmund. 2023. Analysing the Impact of Workloads on Modeling the Performance of Configurable Software Systems. In *Proceedings of ICSE 2023*. 2085–2097.

[68] Changhai Nie and Hareton Leung. 2011. A Survey of Combinatorial Testing. *Comput. Surveys* 43, 2 (2011), 11:1–11:29.

[69] Jeho Oh, Paul Gazzillo, and Don S. Batory. 2019. *t*-wise Coverage by Uniform Sampling. In *Proceedings of SPLC 2019*. 15:1–15:4.

[70] Rongqi Pan, Taher A. Ghaleb, and Lionel Briand. 2023. ATM: Black-box Test Case Minimization based on Test Code Similarity and Evolutionary Search. In *Proceedings of ICSE 2023*. 1700–1711.

[71] Mingyu Park, Hoon Jang, Taejoon Byun, and Yunja Choi. 2020. Property-based testing for LG home appliances using accelerated software-in-the-loop simulation. In *Proceedings of ICSE-SEIP 2020*. 120–129.

[72] Justyna Petke, Myra B. Cohen, Mark Harman, and Shin Yoo. 2015. Practical Combinatorial Interaction Testing: Empirical Findings on Efficiency and Early Fault Detection. *IEEE Transactions on Software Engineering* 41, 9 (2015), 901–924.

[73] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of ESEC/FSE 2013*. 26–36.

[74] 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 SPLC 2019*. 14:1–14:6.

[75] 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 *Proceedings of ICST 2019*. 240–251.

[76] Steven D. Prestwich. 2021. CNF Encodings. In *Handbook of Satisfiability - Second Edition*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 75–100.

[77] Bo Qiao, Fangkai Yang, Chuan Luo, Yanan Wang, Johnny Li, Qingwei Lin, Hongyu Zhang, Mohit Datta, Andrew Zhou, Thomas Moscibroda, Saravanakumar Rajmohan, and Dongmei Zhang. 2021. Intelligent Container Reallocation at Microsoft 365. In *Proceedings of ESEC/FSE 2021*. 1438–1443.

[78] Federica Sarro, Mark Harman, Yue Jia, and Yuanyuan Zhang. 2018. Customer Rating Reactions Can Be Predicted Purely using App Features. In *Proceedings of RE 2018*. 76–87.

[79] Jing Sun, Hongyu Zhang, Yuan-Fang Li, and Hai H. Wang. 2005. Formal Semantics and Verification for Feature Modeling. In *Proceedings of ICECCS 2005*. 303–312.

[80] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* 47, 1 (2014), 6:1–6:45.

[81] Yu-Wen Tung and Wafa S. Aldiwan. 2000. Automating Test Case Generation for the New Generation Mission Software System. In *Proceedings of IEEE Aerospace Conference 2000*. 431–437.

[82] Hiroshi Ukai, Xiao Qu, Hironori Washizaki, and Yoshiaki Fukazawa. 2019. Reduce Test Cost by Reusing Test Oracles through Combinatorial Join. In *Proceedngs of ICSTW 2019*. 260–263.

[83] Hiroshi Ukai, Xiao Qu, Hironori Washizaki, and Yoshiaki Fukazawa. 2022. Accelerating Covering Array Generation by Combinatorial Join for Industry Scale Software Testing. *PeerJ Computer Science* 8 (2022), e720.

[84] 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.

[85] Ziyuan Wang, Changhai Nie, and Baowen Xu. 2007. Generating Combinatorial Test Suite for Interaction Relationship. In *Proceedings of SOQUA 2007*. 55–61.

[86] Max Weber, Christian Kaltenecker, Florian Sattler, Sven Apel, and Norbert Siegmund. 2023. Twins or False Friends? A Study on Energy Consumption and Performance of Configurable Software. In *Proceedings of ICSE 2023*. 2098–2110.

[87] Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. 2020. An Empirical Comparison of Combinatorial Testing, Random Testing and Adaptive Random Testing. *IEEE Transactions on Software Engineering* 46, 3 (2020), 302–320.

[88] Huayao Wu, Changhai Nie, Justyna Petke, Yue Jia, and Mark Harman. 2021. Comparative Analysis of Constraint Handling Techniques for Constrained Combinatorial Testing. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2549–2562.

[89] 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.

[90] Yi Xiang, Han Huang, Sizhe Li, Miqing Li, Chuan Luo, and Xiaowei Yang. 2024. Automated Test Suite Generation for Software Product Lines Based on Quality-Diversity Optimization. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2024), 46:1–46:52.

[91] Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. 2016. Greedy Combinatorial Test Case Generation using Unsatisfiable Cores. In *Proceedings of ASE 2016*. 614–624.

[92] Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. 2015. Optimization of Combinatorial Testing by Incremental SAT Solving. In *Proceedings of ICST 2015*. 1–10.

[93] Cemal Yilmaz, Myra B. Cohen, and Adam A. Porter. 2006. Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces. *IEEE Transactions on Software Engineering* 32, 1 (2006), 20–34.

[94] Linbin Yu, Yu Lei, Mehra Nouroz Borazjany, Raghu Kacker, and D. Richard Kuhn. 2013. An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation. In *Proceedings of ICST 2013*. 242–251.

[95] Jian Zhang, Zhiqiang Zhang, and Feifei Ma. 2014. *Automatic Generation of Combinatorial Test Data*. Springer.

[96] Zhiqiang Zhang, Jun Yan, Yong Zhao, and Jian Zhang. 2014. Generating Combinatorial Test Suite using Combinatorial Optimization. *Journal of Systems and Software* 98 (2014), 191–207.

[97] 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 ESEC/FSE 2023*. 81–93.

Received 2024-04-12; accepted 2024-07-03