SamplingCA: Effective and Efficient Sampling-Based Pairwise Testing for Highly Configurable Software Systems

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

Qiyuan Zhao
Shanghai Jiao Tong University
Shanghai, China
zqy1018@sjtu.edu.cn

Shaowei Cai
Institute of Software, Chinese Academy of Sciences
Beijing, China
caisw@ios.ac.cn

Hongyu Zhang
The University of Newcastle
Callaghan, Australia
hongyu.zhang@newcastle.edu.au

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

ABSTRACT

Combinatorial interaction testing (CIT) is an effective paradigm for testing highly configurable systems, and its goal is to generate a t-wise covering array (CA) as a test suite, where t is the strength of testing. It is recognized that pairwise testing (*i.e.,* CIT with t=2) is the most common CIT technique, and has high fault detection capability in practice. The problem of pairwise CA generation (PCAG), which is a core problem in pairwise testing, aims at generating a pairwise CA (*i.e.,* 2-wise CA) of minimum size, subject to hard constraints. The PCAG problem is a hard combinatorial optimization problem, which urgently requires practical methods for generating pairwise CAs (PCAs) of small sizes. However, existing PCAG algorithms suffer from the severe scalability issue; that is, when solving large-scale PCAG instances, existing state-of-the-art PCAG algorithms usually cost a fairly long time to generate large PCAs, which would make the testing of highly configurable systems both ineffective and inefficient. In this paper, we propose a novel and effective sampling-based approach dubbed *SamplingCA* for solving the PCAG problem. *SamplingCA* first utilizes sampling techniques to obtain a small test suite that covers valid pairwise tuples as many as possible, and then adds a few more test cases into the test suite to ensure that all valid pairwise tuples are covered. Extensive experiments on 125 public PCAG instances show that our approach can generate much smaller PCAs than its state-of-the-art competitors, indicating the effectiveness of *SamplingCA*. Also, our experiments show that *SamplingCA* runs one to two orders of magnitude faster than its competitors, demonstrating the efficiency of *SamplingCA*. Our results confirm that *SamplingCA* is able to address the scalability issue and considerably pushes forward the state of the art in PCAG solving.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Search-based software engineering*.

KEYWORDS

Pairwise Testing, Covering Array, Sampling, Satisfiability

ACM Reference Format:
Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. 2022. *SamplingCA*: Effective and Efficient Sampling-Based Pairwise Testing for Highly Configurable Software Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549155

1 INTRODUCTION

Highly configurable software systems have been instrumental in satisfying increasing demands on customizing software and services [4, 28, 40, 67]. A typical highly configurable system provides many configuration options for users to configure the system [1, 59]. However, it is recognized that testing such a highly configurable system can be challenging [49, 54]. Although generally, a few configurations (*i.e.,* settings of all options) would break the system, the number of all possible configurations grows exponentially as the number of options increases, making it impractical to test all of them. As an example, for a configurable system with 50 configuration options, where each option can take 2 possible values, the number of all possible configurations is $2^{50} = 1,125,899,906,842,624$ (more than 10^{15}) in the worst case.

Combinatorial interaction testing (CIT) is a popular and effective testing paradigm to discover the faults triggered by the interactions of t options, where t is the strength of testing [49, 61, 82]. It is well acknowledged that pairwise testing (CIT with $t = 2$) is the most common CIT technique in practice, since pairwise testing is able to build a small test suite while exhibiting high fault detection ability in real-world applications [11, 25, 74, 83]. Actually, empirical studies [31, 32] on extensive real-world configurable systems reveal that most faults can be detected by pairwise testing, indicating the effectiveness of pairwise testing in practice.

Pairwise testing constructs a reasonable number of test cases (*i.e.,* configurations), where each test case is sampled from the entire

[*] Corresponding author.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

configuration space, and thus a considerable amount of testing effort can be saved. Given a configurable system, a pairwise tuple is a combination of values of two options, and the primary goal of pairwise testing is to generate a pairwise covering array (PCA), which is a set of test cases, such that all possible pairwise tuples are covered. For real-world configurable systems, there are also hard constraints (*e.g.,* functional dependencies and exclusiveness) over configuration options. Since neglecting these hard constraints would lead to inaccurate testing results and wasting testing budget, each test case in a generated PCA must satisfy all hard constraints. Thus, the problem of pairwise covering array generation (PCAG), which aims to build a PCA of minimum size while satisfying all hard constraints, is a fundamental problem in pairwise testing. Solving PCAG is challenging, since the PCAG problem is a hard combinatorial optimization problem [35, 64].

There are three major classes of practical PCAG algorithms: constraint-encoding algorithms (*e.g.,* [3, 26, 81, 87]), greedy algorithms (*e.g.,* [7–9, 11, 33–35, 76, 78, 80]) and meta-heuristic algorithms (*e.g.,* [7, 12–14, 20, 22, 23, 27, 39, 40, 49, 58]). Constraint-encoding algorithms first encode given PCAG instances as the instances of other combinatorial optimization problems, and then call existing optimization solvers to handle such encoded instances. Nevertheless, constraint-encoding algorithms can only deal with small PCAG instances. Greedy algorithms can handle PCAG instances of medium scale, but the PCAs generated by them are usually of large size. Thus, greedy algorithms are infeasible in those real-world application scenarios where testing a single test case would cost much computation resources. Apart from constraint-encoding algorithms and greedy algorithms, meta-heuristic algorithms can generate much smaller PCAs. However, existing PCAG algorithms (including constraint-encoding ones, greedy ones and meta-heuristic ones) suffer from the severe scalability issue [68, 79]. When solving a large PCAG instance, they cost a fairly long time to generate a PCA of large size. Thus, using such PCA would incur ineffective and inefficient testing of highly configurable systems.

Different from PCAG algorithms, a considerable amount of effort has been paid on developing sampling approaches (*e.g.,* [4, 10, 17, 41, 54, 63, 65, 66, 69]), which can rapidly construct valid test cases for a configurable system with many options. However, recent studies [4, 54, 66] indicate that, even if a sampling approach generates plenty of test cases (*e.g.,* thousands of test cases), such huge-sized test suite is still unable to cover all valid pairwise tuples. Hence, testing a configurable system with such test suite would possibly fail to detect a certain number of faults.

In this work, we are devoted to designing an effective and efficient approach to address the scalability issue and thus to push forward the state of the art in PCAG solving. Particularly, we propose an effective and efficient sampling-based approach dubbed *SamplingCA* for solving the PCAG problem. After the initialization steps, *SamplingCA* first works in the sampling phase, and then works in the full covering phase. In the sampling phase, *SamplingCA* uses advanced sampling techniques to construct a small test suite T that covers valid pairwise tuples as many as possible. Thanks to the efficiency characteristic of sampling techniques, the construction process of T in the sampling phase is efficient. Notably, in the sampling phase, *SamplingCA* conducts an iterative process: in each iteration, *SamplingCA* first samples a set of multiple test cases that are dissimilar with respect to the test cases in T; then from the test case set *SamplingCA* selects the one β^* such that $T \cup \{\beta^*\}$ covers the most valid pairwise tuples, and adds β^* into T. In practice, through iteratively selecting test case in a greedy manner, a small test suite with high coverage can be generated. Further, we propose two novel core techniques, *i.e.,* context-aware Boolean satisfiability (SAT) algorithm and variable order randomization strategy, to strengthen the effectiveness of the sampling phase. In the full covering phase, *SamplingCA* further adds a few test cases into T to guarantee that T achieves the full coverage, which makes T become a PCA. Hence, through both sampling phase and full covering phase, *SamplingCA* can solve the PCAG problem both efficiently and effectively.

Extensive experiments, on 125 public instances collected from real-world configurable systems, show that *SamplingCA* can generate much smaller PCAs than all its state-of-the-art competitors (including *AutoCCAG* [49], *FastCA* [38, 39] and *TCA* [40]), indicating the effectiveness of *SamplingCA*. Also, *SamplingCA* runs one to two orders of magnitude faster than all its competitors, demonstrating the efficiency of *SamplingCA*. More encouragingly, on a huge instance (*i.e.,* `uClinux-config`) with 11,254 options and 31,637 hard constraints, *SamplingCA* can generate a PCA of less than 70 test cases, while all its competitors fail to generate PCAs. The results indicate that *SamplingCA* can address the scalability issue and is able to significantly advance the state of the art in PCAG solving.

We summarize our main contributions as below.

- We propose a novel and effective sampling-based algorithm dubbed *SamplingCA*, which can address the scalability issue and advance the state of the art in PCAG solving.
- *SamplingCA* incorporates a number of core techniques, including context-aware SAT algorithm and variable order randomization strategy, to enhance its effectiveness.
- We perform extensive experiments to evaluate *SamplingCA*. The experimental results present that *SamplingCA* is much more effective and efficient than its competitors, indicating that *SamplingCA* might bring practical benefits.

The remainder of this paper is structured as follows. In Section 2, we provide necessary preliminaries about pairwise testing, the PCAG problem, Boolean formulae and a practical algorithm for handling Boolean formulae. In Section 3, we propose the *SamplingCA* approach, and present the core algorithmic techniques of *SamplingCA* in detail. In Section 4, we introduce the experimental design. In Section 5, we perform extensive experiments to evaluate the effectiveness of *SamplingCA*, and report the experimental results. In Section 6, we briefly review the related work. In Section 7, we conclude this paper.

2 PRELIMINARIES

In this section, we introduce necessary preliminaries of this work. We first introduce pairwise testing and the PCAG problem. Then, we describe the relationship between configurable systems and Boolean formulae, and present a practical algorithm called *DPLL* [16] for dealing with Boolean formulae, which is important in *SamplingCA*.

2.1 Pairwise Testing and the PCAG Problem

We describe necessary notations about pairwise testing as follows.

System Under Test: A *system under test* (SUT, also known as configurable system) can be regarded as a pair $S = (O, H)$, where O is a set of *option*s, and H is a collection of *hard constraint*s on the feasible combinations of values of options in O. For each option $o_i \in O$, we use notation R_i to denote the set of all possible values o_i can take.

Without loss of generality, following recent studies on testing highly configurable systems [4, 54], this work studies the binary scenario where each option takes binary values; that is to say, in this work, for each option $o_i \in O$, R_i is $\{0, 1\}$. In fact, it is recognized that the general scenario, where each option takes multiple values, can be effectively modeled as a binary scenario [4, 54]. Actually, all benchmarking instances adopted in this work are encoded from the general scenario and collected from real-world highly configurable systems. Hence, analyzing the binary scenario is of great importance in practice.

Pairwise Tuple: Given an SUT $S = (O, H)$, a *pairwise tuple* is a collection of two pairs, *i.e.,* $\tau = \{(o_{i_1}, r_{i_1}), (o_{i_2}, r_{i_2})\}$, indicating that option $o_{i_1} \in O$ takes value $r_{i_1} \in R_{i_1}$, and meanwhile option $o_{i_2} \in O$ takes value $r_{i_2} \in R_{i_2}$.

Test Case: Given an SUT $S = (O, H)$, a *test case* (also known as *configuration*) covers all options in O, and thus is a set of $|O|$ pairs, *i.e.,* $tc = \{(o_1, r_1), (o_2, r_2), \dots, (o_{|O|}, r_{|O|})\}$, implying that option $o_i \in O$ takes value $r_i \in R_i$. In fact, a test case can be considered as a complete assignment to the set of all options O.

A pairwise tuple τ is *covered* by a test case tc if $\tau \subseteq tc$, implying that each option in τ takes the same value as that in tc. Also, a pairwise tuple τ is covered by a test suite (*i.e.,* a set of test cases) if τ is covered by at least one test case in the test suite.

Many highly configurable systems have hard constraints on the combinations of values of options. Testing highly configurable systems with invalid test cases (*i.e.,* test cases violating any hard constraint) would waste much testing budget, and might lead to incorrect testing results. Hence, it is crucial that each generated test case satisfies all hard constraints. Given an SUT $S = (O, H)$, a test case tc is *valid* if tc satisfies all hard constraints in H. A pairwise tuple τ is *valid* if τ is covered by at least one valid test case.

Pairwise Covering Array: Given an SUT $S = (O, H)$, a *pairwise covering array* (PCA) is a collection of valid test cases, denoted by T, which ensures that each valid pairwise tuple is covered by at least one test case in T.

The problem of *pairwise covering array generation* (PCAG) is to seek a PCA of minimum size, which is the core problem of pairwise testing. The PCAG problem is a hard combinatorial optimization problem [35, 64], so solving PCAG is quite challenging, which urgently calls for effective and efficient methods.

2.2 Boolean Formulae

It is well acknowledged that an SUT is able to be modeled as a Boolean formula [2, 5, 59, 73], and using techniques for dealing with Boolean formulae is known to be an effective way to handle highly configurable systems [37, 60]. Therefore, here we introduce Boolean formulae and discuss the relationship between highly configurable systems and Boolean formulae.

For a Boolean *variable* x, a *literal* of variable x is either itself (x) or its negation ($\neg x$), and a *clause* c is a disjunction of literals. Given n Boolean variables, a Boolean *formula* F in conjunctive normal form (CNF) is a conjunction of m clauses, *i.e.,* $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$, where c_i $(1 \leq i \leq m)$ is a clause; we use notation $V(F)$ to denote the set of all Boolean variables in F, and notation $C(F)$ to represent the set of all clauses in F.

Algorithm 1: *DPLL* Algorithm

```
Input: $F$: Boolean formula in CNF;
       $\alpha$: (partial) assignment of $F$;
Output: solution of $F$, or reporting "No Solution";
if $\alpha$ is a solution then return $\alpha$;
if No solution can be extended from $\alpha$ then return "No Solution";
$x \leftarrow$ the first unassigned variable in $V(F)$ by a fixed variable order;
$D \leftarrow$ a fixed ordered list containing 1 and 0, i.e., $D = [1, 0]$;
foreach $v$ in $D$ do
    $\alpha[x] \leftarrow v$;
    $F', \alpha' \leftarrow$ Simplify $F$ and extend $\alpha$ through unit propagation;
    if DPLL$(F', \alpha')$ returns a solution $\beta$ then return $\beta$;
return "No Solution";
```

For a Boolean variable x, its *truth value* $\nu(x)$ can be assigned either 0 or 1, and the values of literals x and $\neg x$ are $\nu(x)$ and $1 - \nu(x)$, respectively. Given a formula F in CNF, an *assignment* of F is a mapping $\alpha : V(F) \rightarrow \{0, 1\}$. If an assignment α maps all variables, α is a *complete assignment*; otherwise, α is a *partial assignment*. Specifically, ϵ denotes an *empty assignment*, where all variables are unassigned. Given an assignment α, if variable x is assigned under α, then $\alpha[x]$ denotes x's assigned value under α. Given a formula F in CNF and its assignment α, a clause $c \in C(F)$ is *satisfied* if at least one literal l in c evaluates to be 1 under α (*i.e.,* l's value is 1 under α); otherwise c is *unsatisfied*. A *solution* or *satisfying assignment* of F is a complete assignment that makes all clauses in $C(F)$ satisfied. If there exists at least one satisfying assignment of F, then F is *satisfiable*; otherwise, F is *unsatisfiable*.

As mentioned before, an SUT can be modeled as a Boolean formula, usually expressed in CNF [2, 5, 59, 73]. Given an SUT $S = (O, H)$ and its modeled formula F, the option set O of S is related to F's variable set $V(F)$, and the hard constraint set H of S corresponds to F's clause set $C(F)$. A valid test case tc of S is actually a solution α of F. Besides, a pairwise tuple of S is a combination of two literals of F; for example, a pairwise tuple $\tau = \{(o_1, 0), (o_2, 1)\}$ corresponds to the combination $\{\neg x_1, x_2\}$ of F. For simplicity, in this work a combination of two literals of F is also called a pairwise tuple of F. In addition, a pairwise tuple τ of F is covered by a solution α of F if each literal in τ evaluates to be 1 under α.

Given an SUT and its corresponding formula F, the PCAG problem is equivalent to finding a set of F's solutions such that all valid pairwise tuples are covered.

2.3 *DPLL* Algorithm

In fact, given a Boolean formula F, the problem of finding a solution of F is known as the well-studied SAT problem [6]. As discussed before, finding a solution of a formula (*i.e.,* solving SAT) is crucial in solving the PCAG problem. Hence, we briefly review the well-known SAT algorithm called *DPLL* [16], which plays a critical role in our *SamplingCA* approach.

The pseudo-code of *DPLL* is outlined in Algorithm 1. Given a Boolean formula F in CNF and a (partial) assignment α of F, *DPLL*(F, α) returns a solution extended from α if it exists, and reports "No Solution" otherwise. A solution β is extended from a (partial) assignment α if $\alpha[x] = \beta[x]$ for each assigned variable x in α. Therefore, recalling that ϵ denotes an empty assignment, *DPLL*(F, ϵ) returns a solution of F if F is satisfiable.

According to Algorithm 1, *DPLL* is a search algorithm based on recursion. Whenever α becomes a solution of F, the entire process of *DPLL* terminates and α is returned as output (Line 1 in Algorithm 1). In each recursion, if no solution of F can be extended from α (*e.g.,* a clause c is unsatisfied while all variables appearing in c are assigned under α), then the current recursion terminates and *DPLL* moves back to the previous recursion (Line 2 in Algorithm 1). Otherwise, *DPLL* tries to extend the current assignment α by assigning a truth value to an unassigned variable, and activates the next recursion to seek a solution of F. In particular, the variable x to be assigned is chosen as the first unassigned variable x according to a variable order[1] (Line 3 in Algorithm 1). Then *DPLL* tries to assign 1 and 0 to x sequentially (Lines 4–8 in Algorithm 1). Once x is assigned a truth value, *DPLL* employs a powerful reasoning technique called unit propagation [85] to simplify the given formula F through extending redundant, unassigned variables and eliminating unnecessary clauses,[2] resulting in a simplified formula F' and an extended assignment α' (Line 7 in Algorithm 1). After unit propagation, *DPLL* would activate the next recursion to deal with F' and α' (Line 8 in Algorithm 1).

Remark: The variable order for selecting the next unassigned variable (Line 3 in Algorithm 1) and the ordered list of truth values to be assigned (Line 4 in Algorithm 1) decide the search direction of *DPLL*, and the search is conducted through recursion (Line 8 in Algorithm 1). These three parts are marked in red color in Algorithm 1. In this way, all assignments extended from α would be tried, until the first solution extended from α is found. For more details about *DPLL*, readers can refer to the literature [16].

3 OUR PROPOSED *SAMPLINGCA* APPROACH

In this section, we propose and describe *SamplingCA*, an effective and efficient approach for solving the PCAG problem. We first present the top-level design of *SamplingCA*, and then describe the core algorithmic techniques of *SamplingCA* in detail.

Algorithm 2: Top-level Design of *SamplingCA*

```
   Input: $F$: Boolean formula in CNF;
   Output: $T$: pairwise covering array (PCA) of $F$;
 1 if DPLL($F, \epsilon$) reports "No Solution" then return $\emptyset$;
 2 $\alpha \leftarrow DPLL(F, \epsilon)$;
 3 $T \leftarrow \{\alpha\}$;
 4 $csprob \leftarrow UpdateCSProb(F, T)$;
 5 while True do
 6     for $j \leftarrow 1$ to $k$ do
 7         $\gamma_j \leftarrow$ Sample a reference assignment according to
             $csprob$;
 8         $\pi_j \leftarrow$ a random variable order of $V(F)$;
 9         $\beta_j \leftarrow ContextSAT(F, \epsilon, \gamma_j, \pi_j)$;
10     $\beta^* \leftarrow \arg\max_{\beta_j} gain(\beta_j, T)$, where $1 \leq j \leq k$;
11     if $gain(\beta^*, T) \leq 0$ then break;
12     $T \leftarrow T \cup \{\beta^*\}$;
13     $csprob \leftarrow UpdateCSProb(F, T)$;
14 foreach possible pairwise tuple $\tau$ of $F$ do
15     if $\tau$ is not covered by any test case in $T$ then
16         if DPLL($F, \tau$) returns a solution $tc$ then
               $T \leftarrow T \cup \{tc\}$;
17 return $T$;
```

3.1 Top-level Design of *SamplingCA*

As described before, existing PCAG algorithms suffer from the scalability issue and require a large amount of computation time to generate PCAs of large sizes when handling highly configurable systems. Compared to existing PCAG algorithms, sampling approaches can efficiently generate valid test cases for highly configurable systems. However, the test suite generated by a sampling approach cannot achieve full coverage (*i.e.,* covering all valid pairwise tuples) even if such test suite contains plenty of valid test cases (*e.g.,* thousands of valid test cases) [4, 54, 66], so it is apparent that sampling approaches cannot generate PCAs.

Inspired by the efficiency characteristic of sampling approaches [4, 54, 66], the main ideas behind *SamplingCA* are as follows: 1) *SamplingCA* utilizes sampling techniques to obtain a small test suite T that covers valid pairwise tuples as many as possible, and then 2) *SamplingCA* adds a few valid test cases into T to ensure that all valid pairwise tuples are covered, which makes T become a PCA. The top-level design of *SamplingCA* is listed in Algorithm 2. *SamplingCA* takes a Boolean formula F that is modeled from an SUT as its input, and returns a PCA dubbed T as its output.

As presented in Algorithm 2, *SamplingCA* consists of three phases, *i.e.,* initialization phase, sampling phase and full covering phase. In the initialization phase, the first valid test case is generated, and necessary initialization steps are conducted (Lines 1–4 in Algorithm 2). In the sampling phase, *SamplingCA* iteratively calls a novel and effective SAT algorithm to achieve a test suite T that covers valid pairwise tuples as many as possible (Lines 5–13 in Algorithm 2). In the full covering phase, *SamplingCA* adds a few valid test cases into T to ensure that T covers all valid pairwise tuples, which makes T become a PCA (Lines 14–16 in Algorithm 2).

Through this way, *SamplingCA* is guaranteed to generate a PCA for a given SUT.

[1] The variable order remains fixed during the entire process of *DPLL*, and different implementations of *DPLL* adopt distinct orders. The most commonly-used order is the one determined by a heuristic called VSIDS (Variable State Independent Decaying Sum) [62]. For more details about VSIDS, readers can refer to literature [62, 72].

[2] For example, a clause c in F has only one literal l, and l's corresponding variable x is still unassigned. Then x is a redundant variable, since x must be assigned the value such that l evaluate to be 1. Finally, α is extended accordingly, and c becomes an unnecessary clause and would be eliminated.

3.2 Challenges

Before introducing the core techniques of *SamplingCA*, here we present two challenges that need to be addressed by *SamplingCA*.

3.2.1 Diversity Challenge. In practice, minimizing the size of PCA is of great importance, since a large PCA would lead to inefficient testing of highly configurable systems. Actually, the size of the test suite generated by the sampling phase directly impacts the size of the final PCA output by *SamplingCA*. Thus, the sampling phase aims to construct a test suite T of small size while covering valid pairwise tuple as many as possible. Also, if T covers more valid pairwise tuples, then there will be fewer uncovered, valid pairwise tuples, which means that the number of new test cases required by T to become a PCA is less. Hence, in the sampling phase, constructing a small test suite with high coverage is desirable.

To achieve this task, an advisable solution is to construct a test suite T with high diversity (*i.e.,* every test case in T has high dissimilarity with each other). The intuition behind this solution is straightforward: a test suite consisting of dissimilar test cases can cover more valid pairwise tuples than the one containing similar test cases. Therefore, failing to effectively handle the diversity challenge could make the solving of the PCAG problem ineffective.

3.2.2 Completeness Challenge. When solving PCAG, many state-of-the-art PCAG algorithms (*e.g., AutoCCAG* [49], *FastCA* [38, 39] and *TCA* [40]) need to obtain the set of all valid pairwise tuples before generating any test case. In particular, such PCAG solvers first enumerate all possible pairwise tuples. Then, for each possible pairwise tuple τ, an SAT algorithm (*e.g., DPLL* [16]) is called to justify whether there is a valid test case covering τ. If this is the case, τ is valid; otherwise, τ is invalid.

Given an SUT with n options, the number of possible pairwise tuples is $\binom{n}{2} \times 4$. That is to say, for many advanced algorithms (*e.g., AutoCCAG*, *FastCA* and *TCA*), to obtain all valid pairwise tuples, the number of SAT algorithm calls is $\binom{n}{2} \times 4$. However, SAT is a computationally hard problem [6]; although modern SAT algorithms can solve large SAT instances, calling an SAT algorithm to find a solution still requires a certain amount of execution time [4]. Therefore, the process of obtaining all valid pairwise tuples used by state-of-the-art PCAG algorithms would cost considerable computation time when n is large (*i.e.,* dealing with a highly configurable system). For example, the highly configurable system `uClinux-config` has 11,254 options, and thus achieving all its valid pairwise tuples requires more than 250 million SAT algorithm calls, which is much time-consuming and even infeasible in real-world applications [4]. Hence, failing to effectively deal with the completeness challenge hinders solving PCAG from being efficient.

3.3 Initialization Phase

In the initialization phase, the primary goal of *SamplingCA* is to generate the first valid test case. As aforementioned, a solution of the given formula F is actually a valid test case, so it is advisable to use an SAT algorithm to obtain a solution of F. In particular, *SamplingCA* calls *DPLL* to achieve the first valid test case α (Line 2 in Algorithm 2).[3]

After the first valid test case α is obtained, *SamplingCA* initializes the test suite T (*i.e.,* the set of valid test cases) as the set containing the first valid test case, *i.e.,* $\{\alpha\}$ (Line 3 in Algorithm 2). We note that, in the sampling phase and the full covering phase, valid test cases would be iteratively added into T until T becomes a PCA. Once T becomes a PCA, it would be returned as the output of *SamplingCA*. In addition, at the end of the initialization phase, based on T, *SamplingCA* initializes the context-aware sampling probability for each variable in F (which plays a key role in the sampling phase and will be explained in Section 3.4.1),

3.4 Sampling Phase

In the sampling phase, *SamplingCA* conducts an iterative process to obtain a test suite, and in each iteration, *SamplingCA* activates a novel and effective SAT algorithm to generate a valid test case. The main target of the sampling phase of *SamplingCA* is to achieve a test suite that covers valid pairwise tuples as many as possible. As discussed in Section 3.2.1, an effective solution is to construct a test suite T with high diversity, *i.e.,* containing a set of dissimilar test cases. Based on this idea, in each iteration, *SamplingCA* aims to generate a valid test case β that can enhance the diversity of T if β is added into T.

To enhance T's diversity, in each iteration, it is intuitive to first generate multiple valid test cases that are dissimilar with respect to the test cases in T, and then from those generated test cases select the one with the highest dissimilarity. Through this way, the test case β^* selected in each iteration can considerably enhance the diversity of T if β^* is added into T.

Following this intuition, how to quantify the dissimilarity of a valid test case with respect to the test cases in T is a core problem that needs to be addressed. Actually, in the sampling phase, an ultimate objective is to make T cover valid pairwise tuples as many as possible. To this end, inspired by the literature [54], we employ a metric called *gain*, to quantify the contribution of a test case over T, and how to calculate the value of gain is described as follows. Given a valid test case β and a test suite T, the gain of β with respect to T, denoted by $gain(\beta, T)$, is the number of valid pairwise tuples that are covered by β but not covered by any test case in T. Clearly, $gain(\beta, T)$ can directly evaluate the improvement on the ultimate objective, which is brought by β over T if β is added into T.

Inspired by the literature [11, 54], in each iteration, *SamplingCA* adopts a greedy selection mechanism to choose the valid test case. The greedy selection mechanism works as follows: First, a candidate set of k valid test cases is constructed; Then, from the entire candidate set the test case β^* with the largest gain is selected. Here k is an integer-valued hyper-parameter that plays a key role in balancing the effectiveness and the efficiency of the sampling phase. On one hand, setting k to a larger value would reduce the size of the generated test suite through examining more candidates while more running time is required. On the other hand, setting k to a smaller value would make *SamplingCA* run faster but achieve a larger test suite. The effect of k will be studied empirically in Section 5.4.

[3] If F is unsatisfiable, which means that there is no solution of F, then *SamplingCA* terminates and returns an empty set as its output (Line 1 in Algorithm 2). We note that this case would not happen in real-world applications, since a normal SUT has at least one valid configuration. We consider this case for the correctness of *SamplingCA* (*i.e.,* ensuring that *SamplingCA* can handle all situations).

Algorithm 3: *UpdateCSProb* Method

```
Input: $F$: Boolean formula in CNF;
       $T$: current, non-empty test suite;
Output: $csprob$: calculated context-aware sampling probabilities;
foreach $x_i \in V(F)$ do
    $\delta \leftarrow$ the number of test cases in $T$, where $x_i$'s value is 1;
    $csprob(x) \leftarrow \delta/|T|$;
return $csprob$;
```

Algorithm 4: *ContextSAT* Algorithm

```
Input: $F$: Boolean formula in CNF;
       $\alpha$: (partial) assignment of $F$;
       $\gamma$: a reference assignment of $V(F)$;
       $\pi$: a variable order of $V(F)$;
Output: solution of $F$, or reporting "No Solution";
1 if $\alpha$ is a solution then return $\alpha$;
2 if No solution can be extended from $\alpha$ then return "No Solution";
3 $x \leftarrow$ the first unassigned variable in $V(F)$ according to $\pi$;
4 $D \leftarrow$ an ordered list based on $\gamma$, i.e., $D = [\gamma[x], 1 - \gamma[x]]$;
5 foreach $v$ in $D$ do
6     $\alpha[x] \leftarrow v$;
7     $F', \alpha' \leftarrow$ Simplify $F$ and extend $\alpha$ through unit propagation;
8     if ContextSAT$(F', \alpha', \gamma, \pi)$ returns a solution $\beta$ then return $\beta$;
9 return "No Solution";
```

In addition, as the iterative process proceeds, more test cases are added into T, so more valid pairwise tuples are covered by T. That is to say, the gains of those test cases selected in subsequent iterations tend to decrease. Once the gain of the selected test case β^* becomes 0 (*i.e.,* each pairwise tuple covered by β^* is also covered by at least one test case in T), the iterative process terminates. Adopting such termination criterion could prevent redundant test cases from being added into T, which prevents the size of generated test suite from being meaninglessly increased. In practice, through iteratively activating the greedy selection mechanism, a small test suite with high coverage can be constructed. The sampling phase is outlined in Lines 5–13 in Algorithm 2.

3.4.1 Context-aware SAT Algorithm. For the sampling phase, in each iteration, a core task is to generate a candidate set of k valid test cases that are dissimilar compared to the test cases in T. As discussed in Section 2.3, to generate a valid test case, a straightforward method is to directly use *DPLL*. However, as an SAT algorithm, *DPLL* concentrates on searching for a solution efficiently, but it does not take the dissimilarity requirement into consideration. This is a severe problem, since neglecting the dissimilarity requirement would negatively impact the diversity of T and thus prevent the number of valid pairwise tuples covered by T from being maximized.

To tackle this problem, we propose a novel and effective context-aware SAT algorithm dubbed *ContextSAT* for constructing valid test cases that are dissimilar compared to the test cases in T. Before describing the technical details of the *ContextSAT* algorithm, inspired by the literature [54], we utilize the concept of context-aware sampling probability, which is described as follows. Given a Boolean formula F, for each variable $x \in V(F)$, the context-aware sampling probability of x, denoted by $csprob[x]$, is the probability that x's value is sampled to be 0. Thus, the probability that x's value is sampled to be 1 is $1 - csprob[x]$. Recent studies present that such sampling approaches where sampling probabilities change dynamically can enhance the diversity of the generated test suite compared to uniform sampling (*i.e.,* for each variable x, $csprob[x]$ remains 0.5 during the entire sampling process) [4, 54].

As a result, in *SamplingCA*, each variable's context-aware sampling probability is computed based on the current context of T, and would be updated once a new test case is added into T (Line 13 in Algorithm 2). In particular, given the current test suite T, to generate a test case that is dissimilar with respect to the test cases in T, it is intuitive that, for each variable x, if the number of test cases in T where x's value is 1, is larger than the number of test cases in T where x's value is 0, then $csprob[x]$ should be larger; otherwise, $csprob[x]$ should be smaller. Based on this intuition, in *SamplingCA*, for a variable x, $csprob[x]$ is calculated as the ratio between the number of test cases in T where x's value is 1, and the number of all test cases in T (*i.e.,* the size of T). The method of calculating *csprob* is outlined in Algorithm 3.

Actually, when generating a valid candidate test case, it is advisable to take *csprob* into consideration, since *csprob* reflects the current context of T. As aforementioned, *DPLL* can generate valid test cases efficiently, but does not consider the dissimilarity requirement. Thus, incorporating *csprob* into *DPLL* can make *DPLL* generate such valid test cases that are dissimilar compared to the test cases in T. Particularly, in each iteration of the sampling phase, when generating the j-th valid candidate test case, a reference assignment γ_j (possibly invalid) is sampled according to *csprob*, where, for each variable x, $\gamma_j[x]$ is sampled based on $csprob[x]$ (Line 7 in Algorithm 2). In this way, γ_j has a high dissimilarity with respect to the test cases in T. Then we design a new SAT algorithm dubbed *ContextSAT*, which is based on *DPLL* and utilizes the reference assignment γ_j to guide the search direction.

The pseudo-code of our *ContextSAT* algorithm is listed in Algorithm 4, where the major differences between *ContextSAT* and *DPLL* are marked in blue color. As discussed in Section 2.3, the variable order for selecting the next unassigned variable, as well as the ordered list of truth values to be assigned, determines the direction of solution search, and the search is conducted based on recursion. Thus, *ContextSAT* introduces two enhancements over *DPLL* on determining both the variable order and the truth value order. The first enhancement will be introduced in Section 3.4.2, and here we introduce the second enhancement. Compared to *DPLL* that always uses a fixed ordered list of truth values, *ContextSAT* determines the ordered list according to the reference assignment γ_j, where the first truth value is the one under γ_j (Line 4 in Algorithm 4). Consequently, based on recursion, *ContextSAT* aims to search for a valid test case β_j, which, as a candidate test case, is similar with γ_j and thus has high dissimilarity with respect to the test cases in T (Line 8 in Algorithm 4).

Through this way, *ContextSAT* can construct a set of candidate test cases, where each candidate can enhance the diversity of T if it is added into T.

3.4.2 Variable Order Randomization Strategy. Here we introduce the enhancement of *ContextSAT* over *DPLL* on determining the variable order. Actually, if *ContextSAT* utilizes the same variable order as *DPLL* does (*e.g.,* the variable order decided by the VSIDS heuristic [62], as described in Section 2.3), then the variable orders adopted by *ContextSAT* are exactly the same when generating multiple valid test cases for the same Boolean formula. That is to say, for each test case in the generated test suite T, it is generated using the same variable order. However, as discussed in Section 2.3, the variable order greatly impacts the search direction, so using the same variable order to generate multiple test cases for T would negatively impact the diversity of T.

In the research field of SAT solving, it is recognized that adopting randomized strategies can enhance diversity [19, 36, 44, 47]. Hence, it is intuitive that integrating *ContextSAT* with randomized strategies can further enhance the diversity of the generated test suite T. To this end, we propose a variable order randomization strategy to determine the variable order in *ContextSAT*. Particularly, when generating a valid test case, *SamplingCA* would first obtain a random variable order of $V(F)$, and then activate *ContextSAT* with the random variable order (Lines 8–9 in Algorithm 2); for *ContextSAT*, it uses the random variable order to select the next unassigned variable (Line 3 in Algorithm 4).

Through the variable order randomization strategy, *ContextSAT* adopts different variable orders for generating various valid test cases, so the diversity of T can be enhanced.

Remark: In the sampling phase, through the context-aware SAT algorithm (Section 3.4.1) and the variable order randomization strategy (Section 3.4.2), *SamplingCA* can generate a small test suite T with high diversity, which can tackle the diversity challenge (Section 3.2.1).

3.5 Full Covering Phase

Once the sampling phase terminates, a test suite T is generated. However, T is not guaranteed to cover all valid pairwise tuples. In the full covering phase, *SamplingCA* aims to add a few valid test cases into T, in order to make T cover all valid pairwise tuples and thus become a PCA. Particularly, in the full covering phase, *SamplingCA* conducts an enumerating process to traverse all possible pairwise tuples. For each possible pairwise tuple τ, *SamplingCA* first checks whether τ is both valid and uncovered; if this is the case, *SamplingCA* adds a valid test case *tc* that covers τ into T. After all valid pairwise tuples are covered by T, T becomes a PCA and is returned by *SamplingCA* as the output.

For the enumerating process, there is an important problem, *i.e.,* how to efficiently check whether a pairwise tuple τ is valid and uncovered. A straightforward method is to 1) first call *DPLL*(F, τ) to justify τ's validity, and 2) then check the covering status of τ. However, given an SUT with n options, there are $\binom{n}{2} \times 4$ possible pairwise tuples, so using the straightforward method needs to call the SAT algorithm $\binom{n}{2} \times 4$ times. To address this problem, we propose an efficient validation method based on the definition of valid pairwise tuple.

As described in Section 2.1, the definition of valid pairwise tuple is as follows: for a pairwise tuple τ, if there exists at least one valid test case covering τ, then τ is a valid pairwise tuple. According to the definition of valid pairwise tuple, since all test cases in T are valid, pairwise tuples that are covered by any test case in T are valid. Hence, we can construct a set of valid pairwise tuples that are covered by at least one test case in T, denoted by M. Then, when checking whether a given pairwise tuple τ is both valid and uncovered, *SamplingCA* first justifies whether τ belongs to M: if so, τ does not satisfy this condition and *SamplingCA* continues to check the next pairwise tuple; otherwise, *SamplingCA* calls *DPLL*(F, τ) to justify τ's validity. Hence, compared to the straightforward method, the number of SAT algorithm calls required by *SamplingCA* is significantly reduced, which makes considerable contribution to its efficiency.

Remark: As discussed in Section 3.2.2, many state-of-the-art PCAG algorithms (*e.g., AutoCCAG*, *FastCA* and *TCA*) need to obtain the set of all valid pairwise tuples before generating any test case, and they call an SAT algorithm one time to validate one single pairwise tuple. Thus, when dealing with an SUT with many options, such existing PCAG algorithms have to activate a large number of SAT algorithm calls, which costs much running time.

Instead, given an SUT with n options, *SamplingCA* does not require obtaining all valid pairwise tuples in advance. In the sampling phase, rather than calling an SAT algorithm to validate one single pairwise tuple, *SamplingCA* calls an SAT algorithm to find a valid test case, which covers $\binom{n}{2}$ valid pairwise tuples. That is to say, through calling SAT algorithm one time, *SamplingCA* can validate a considerable number of pairwise tuples. Hence, compared to many state-of-the-art PCAG algorithms, the number of SAT algorithm calls is obviously reduced, so the efficiency can be significantly improved. Also, in the full covering phase, all valid pairwise tuples are ensured to be covered, which guarantees that *SamplingCA* can generate a PCA. Therefore, *SamplingCA* can generate PCAs for large PCAG instances, indicating that the completeness challenge can be addressed (Section 3.2.2).

4 EXPERIMENTAL DESIGN

In this section, we present our design of the experiments in this work. We first describe the benchmarking instances used in our experiments. Then we introduce the state-of-the-art competitors of *SamplingCA*. Subsequently, we list the research questions, and finally we introduce the experiment setup.

4.1 Benchmarking Instances

In our experiments, we adopt an instance set of 125 PCAG benchmarking instances, all of which are encoded from real-world, highly configurable systems and modeled as Boolean formulae in CNF. These PCAG instances are originally collected by Baranov *et al.* [4], and have been extensively studied in recent works [4, 29, 37, 54, 66, 68, 69]. For all instances adopted in this work, the numbers of options of modeled formulae vary from 94 to 11,254, and the numbers of constraints of modeled formulae range from 190 to 62,183. All instances are publicly available online.[4]

4.2 State-of-the-art Competitors

In this work, *SamplingCA* is compared against three state-of-the-art competitors, *i.e., AutoCCAG* [49], *FastCA* [38, 39] and *TCA* [40].

AutoCCAG [49] is a recent, state-of-the-art algorithm that uses advanced automated optimization techniques. The experimental results in the literature [49] demonstrates that *AutoCCAG* achieves significant performance improvement over *TCA*, *CASA* [20, 21] and *CHiP* [61] on a broad range of practical instances, indicating the effectiveness and the robustness of *AutoCCAG*.

FastCA [38, 39] is an both efficient and effective meta-heuristic algorithm. Reported in the literature [39], *FastCA* performs considerably better and runs much faster than *TCA*, *CASA*, *ACTS* [84] and *HHSA* [27] on a diverse set of real-world instances.

TCA [40] is a high-performance two-mode meta-heuristic algorithm. According to the experiments in the literature [40], *TCA* can generate much smaller PCAs than *CASA*, *ACTS* and *Cascade* [87] on many application instances.

For *AutoCCAG*, the source code is kindly provided by its authors [49]. For the other two competitors, the source codes of *FastCA*[5] and *TCA*[6] are publicly available. For all these 3 competitors (*i.e., AutoCCAG*, *FastCA* and *TCA*), they are evaluated using the hyper-parameter settings recommended in the literature [39, 40, 49], respectively. Besides *AutoCCAG*, *FastCA* and *TCA*, we also evaluate the practical performance of other three well-known algorithms, including *CASA* [20, 21], *HHSA* [27] and *ACTS* [84]. However, our experiments present that *CASA*, *HHSA* and *ACTS* cannot generate PCAs for the majority of the benchmarking instances within the cutoff time that is used in this work and will be described in Section 4.4. Thus, to save space, we do not report the results of *CASA*, *HHSA* and *ACTS* in this paper. The results of comparing *SamplingCA* with *CASA*, *HHSA* and *ACTS* are publicly available online.[4]

4.3 Research Questions

In practice, it is desirable to generate a PCA of small size in a short time for testing a highly configurable system. Hence, for solving PCAG, the size of PCA and the running time are two critical metrics. In our experiments we concentrate on pushing forward the current state of the art in minimizing the size of generated PCA and shortening the running time. Our evaluation of *SamplingCA* aims at answering the following research questions (RQs):

RQ1: Can *SamplingCA* generate smaller PCA compared to its state-of-the-art competitors?

In this RQ, we conduct experiments to compare the size of PCA generated by *SamplingCA* against those of PCAs generated by its 3 state-of-the-art competitors, *i.e., AutoCCAG*, *FastCA* and *TCA*.

RQ2: Can *SamplingCA* generate PCAs more efficiently than its state-of-the-art competitors?

In this RQ, we evaluate the running time of *SamplingCA* for generating PCAs, and compare the running time of *SamplingCA* with that of *AutoCCAG*, *FastCA* and *TCA*.

RQ3: Does each core technique of *SamplingCA* contribute to the performance improvement?

In this RQ, we conduct ablation study to analyze the contribution of the core techniques of *SamplingCA*, including context-aware SAT algorithm and variable order randomization strategy, to the performance improvement. Besides, since the sampling phase of *SamplingCA* is of great importance, we also study the effectiveness of the sampling phase.

RQ4: What impact does the hyper-parameter setting have on the performance of *SamplingCA*?

In this RQ, we study how the setting of hyper-parameter k (introduced in Section 3.4) impacts the performance of *SamplingCA*.

4.4 Experimental Setup

All experiments in this work were performed on a computing machine with Intel Xeon Platinum 8272CL CPU and 256GB memory, running Ubuntu 18.04. We note that *SamplingCA* is implemented on the basis of *LS-Sampling* [54]. Since *LS-Sampling* adopts *Coprocessor* [57] to simplify input Boolean formulae before constructing test suites, *SamplingCA* also employs *Coprocessor* [57] to simplify input Boolean formulae before generating PCAs. The implementation of *SamplingCA* is publicly available online.[4]

SamplingCA and its three state-of-the-art competitors are all randomized approaches, so we performed 100 independent runs per benchmarking instance for each competing approach, with a cutoff time of 3600 CPU seconds per run, following a recent study on solving a hard combinatorial optimization problem [48]. In our experiments, for *SamplingCA*, the hyper-parameter k is set to 100, and the effects of different settings of hyper-parameter k are discussed in Section 5.4. Also, since the influential SAT solver *MiniSAT* [18] provides an efficient implementation of *DPLL*, we use *MiniSAT* as the implementation of *DPLL* in this work, and our *ContextSAT* algorithm is also implemented on top of *MiniSAT*.

For each competing approach on each benchmarking instance, we demonstrate the smallest size of the generated PCAs among 100 runs, denoted by 'min', and the average size of the generated PCAs over 100 runs, denoted by 'avg'. Also, for each approach on each instance, we also report the average running time over 100 runs, denoted by 'time'. In this work, all running times are measured in CPU second. If a competing approach failed to generate a PCA among all 100 runs for a benchmarking instance, we mark the corresponding results of 'min', 'avg' and 'time' as '–'. Besides, for each competing approach (in Tables 2, 3 and 4), we also present the average size and the average time over the entire instance set except two instances (*i.e.,* `embtoolkit` and `uClinux-config`), because all competitors (including *AutoCCAG*, *FastCA* and *TCA*) and a number of *SamplingCA*'s alternative versions are not able to generate PCAs for these two instances among all 100 runs. In our experiments, for each instance (in Table 1) or the entire instance set except `embtoolkit` and `uClinux-config` (in Tables 2, 3 and 4), we utilize **boldface** to indicate the best results. In addition, for each instance (in Table 1), we also report the number of options of the modeled formula and the number of constraints of the modeled formula, denoted by '#Options' and '#Constraints', respectively.

Following the recent study [49], in our experiments, for each instance (in Table 1) or the entire instance set except `embtoolkit`

[4] https://github.com/chuanluocs/SamplingCA
[5] https://github.com/jkunlin/fastca
[6] https://github.com/jkunlin/TCA

Table 1: Comparing *SamplingCA* against *AutoCCAG*, *FastCA* and *TCA* on 20 selected instances.

Instances	#Options	#Constraints	*SamplingCA*		*AutoCCAG*		*FastCA*		*TCA*	
			min (avg)	time	min (avg) size	time	min (avg)	time	min (avg)	time
`adderII`	1276	3206	**101 (109.16)**	**33.37**	166 (183.67)	1605.72	164 (183.82)	1266.56	203 (219.82)	948.99
`at91sam7sek`	1296	3921	**98 (104.10)**	**32.96**	154 (170.23)	1484.51	156 (170.25)	1482.87	187 (204.13)	958.29
`busybox_1_28_0`	998	962	**54 (58.12)**	**11.57**	95 (111.18)	611.19	95 (111.37)	567.30	125 (137.77)	512.78
`dreamcast`	1252	3168	**108 (113.52)**	**33.41**	172 (188.47)	1589.67	172 (188.75)	1195.18	199 (226.71)	918.31
`e7t`	1266	3816	**103 (110.17)**	**32.87**	163 (177.60)	1732.23	163 (177.60)	1543.14	200 (218.47)	937.28
`ecos-icse11`	1244	3146	**103 (108.22)**	**31.04**	260 (286.70)	1662.22	260 (287.23)	1962.32	328 (366.92)	927.75
`embtoolkit`	2128	15483	**1107 (1119.48)**	**891.94**	– (–)	–	– (–)	–	– (–)	–
`financial`	771	7241	**4421 (4433.91)**	**544.31**	5292 (5351.01)	2951.96	5292 (5351.04)	2904.89	5293 (5352.04)	2743.31
`freebsd-icse11`	1396	62183	**116 (121.33)**	**43.94**	164 (184.02)	3478.97	173 (188.62)	3552.38	211 (232.44)	2821.50
`h8max`	1202	3072	**102 (108.07)**	**29.84**	163 (182.41)	1496.94	164 (182.80)	1178.40	192 (219.10)	801.22
`innovator`	1256	50452	**134 (140.95)**	**41.17**	204 (225.73)	2454.67	206 (225.92)	2988.04	255 (270.43)	1556.81
`integrator_arm9`	1267	3939	**116 (122.19)**	**36.94**	197 (213.56)	2445.35	196 (213.70)	3143.44	233 (256.05)	1682.01
`mpc50`	1213	3728	**96 (100.97)**	**28.20**	141 (160.01)	1260.33	145 (160.15)	1220.16	177 (193.85)	819.19
`ocelot`	1266	3141	**103 (107.63)**	**32.19**	163 (181.75)	1618.54	165 (181.87)	1274.91	198 (221.31)	926.08
`pc_i82544`	1259	3179	**106 (111.18)**	**33.15**	169 (188.16)	1561.15	168 (188.65)	1189.46	204 (225.70)	940.95
`ref4955`	1218	3099	**95 (102.02)**	**28.91**	154 (168.99)	1548.75	155 (169.17)	1211.88	185 (202.64)	827.63
`refidt334`	1263	3140	**108 (114.60)**	**34.47**	173 (192.91)	1540.53	173 (193.23)	1324.82	217 (234.44)	985.29
`se7751`	1295	3254	**113 (118.08)**	**36.94**	182 (198.69)	1704.43	181 (198.72)	1396.61	213 (238.97)	1028.23
`uClinux-config`	11254	31637	**63 (66.45)**	**1498.42**	– (–)	–	– (–)	–	– (–)	–
`XSEngine`	1260	3803	**102 (107.95)**	**32.50**	159 (177.14)	1537.39	161 (177.30)	1389.57	191 (214.61)	926.49

Table 2: Average size and average time of *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA* over the entire instance set except two instances (*i.e.,* `embtoolkit` and `uClinux-config`).

	SamplingCA	*AutoCCAG*	*FastCA*	*TCA*
average size	**140.16**	215.64	215.90	251.03
average time	**34.93**	1520.45	1405.08	944.39

and `uClinux-config` (in Tables 2 and 3), we separately compare the sizes of all PCAs generated by *SamplingCA* with those of all PCAs generated by each competitor. In particular, for any comparison between *SamplingCA* and each competitor, we perform the Wilcoxon signed-rank test [15] to examine the statistical significance, and calculate the Vargha-Delaney effect size [77]. For each instance (in Table 1) or the entire instance set except `embtoolkit` and `uClinux-config` (in Tables 2 and 3), if 1) all the p-values of Wilcoxon signed-rank tests at 95% confidence level are smaller than 0.05 (indicating statistical significance) [15, 49], and 2) the Vargha-Delaney effect sizes of all comparisons are larger than 0.71 (indicating large effect sizes) [49, 71, 77], the performance improvement of *SamplingCA* over all its competitors is considered to be statistically significant and meaningful, and the results of *SamplingCA* are marked using underline.

5 EXPERIMENTAL RESULTS

In this section, we report and discuss the experimental results, in order to demonstrate that *SamplingCA* is both effective and efficient.

5.1 Comparison on the Size of PCA (RQ1)

Table 1 presents the comparative results of *SamplingCA* and its three state-of-the-art competitors (*i.e., AutoCCAG*, *FastCA* and *TCA*) on 20 selected instances, where 10 instances are recognized as representative ones in the literature [4], and other 10 instances are randomly chosen. Due to the space limit, we do not report the results on all instances in Table 1. Nevertheless, the full results of *SamplingCA* and all its competitors on all instances are publicly available online.[4] Also, Table 2 summarizes the average size and the average running time of *SamplingCA* and all its competitors over the entire instance set except `embtoolkit` and `uClinux-config`.

According to Tables 1 and 2, *SamplingCA* stands out as the best approach, and it can generate much smaller PCAs than all its competitors. More encouragingly, on the `uClinux-config` instance with 11,254 options and 31,637 constraints, *SamplingCA* can generate a PCA consisting of less than 70 test cases, while all its competitors fail to generate PCAs. The experimental results in Tables 1 and 2 clearly indicate that *SamplingCA* advances the state of the art in minimizing the size of PCA.

5.2 Comparison on Efficiency (RQ2)

The running time required by *SamplingCA* and all its competitors on each selected instance is presented in Table 1. Also, the average time required by these competing approaches over the entire instance set except `embtoolkit` and `uClinux-config` is summarized in Table 2. From Tables 1 and 2, *SamplingCA* runs much faster than all its competitors. Particularly, Table 2 shows that the average time required by *SamplingCA* is one to two orders of magnitude less than all its competitors, indicating the efficiency of *SamplingCA*.

As discussed in Section 3.2.2, when solving an instance, the number of SAT algorithm calls required by each of *AutoCCAG*, *FastCA* and *TCA* is equal to the number of possible pairwise tuples, while *SamplingCA* can reduce the number of SAT algorithm calls. We illustrate the number of *SamplingCA*'s SAT algorithm calls and the number of possible pairwise tuples on each instance in Figure 1,

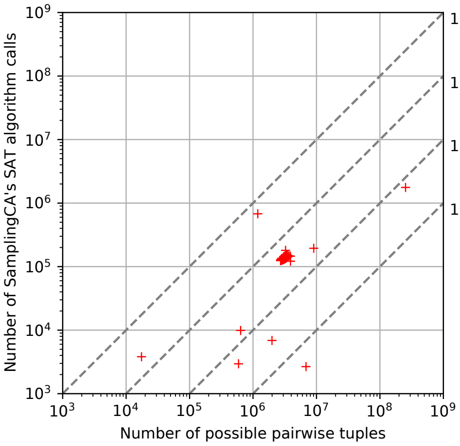

Figure 1: Number of *SamplingCA*'s SAT algorithm calls (in Y-axis) and number of possible pairwise tuples (in X-axis) for all benchmarking instances. A notation '+' in red color represents an instance.

where a notation '+' in red color represents an instance. For the majority of instances, the number of *SamplingCA*'s SAT algorithm calls is significantly less than the number of possible pairwise tuples. As an example, for the `uClinux-config` instance, *SamplingCA* needs to call SAT algorithm 1,764,977 times, while the number of its possible pairwise tuples is 253,282,524; this might be the possible reason why *SamplingCA* can generate a PCA for `uClinux-config` while all its competitors fail to handle `uClinux-config`. Hence, through considerably reducing the number of SAT algorithm calls, *SamplingCA* is much more efficient than all its competitors.

5.3 Effects of Core Techniques (RQ3)

In this subsection, we conduct ablation study to analyze the effects of core techniques of *SamplingCA*. The sampling phase of *SamplingCA* is important, so we modify *SamplingCA* by replacing its sampling phase with a recent, state-of-the-art local search based sampling method named *LS-Sampling* [54], resulting in an alternative version of *SamplingCA* called *Alt-1*. Besides, both context-aware SAT algorithm and variable order randomization strategy are core techniques of *SamplingCA*, so we remove each of them from *SamplingCA*, resulting in two alternative versions named *Alt-2* and *Alt-3*, respectively. The average size and the average time of *SamplingCA* and all its alternative versions are reported in Table 3. In terms of both metrics (*i.e.,* average size and average time), *SamplingCA* significantly outperforms all its alternative versions, indicating the effectiveness of each core technique of *SamplingCA*.

5.4 Impact of Hyper-parameter Setting (RQ4)

In Table 4, we present the average size and the average time of *SamplingCA* with different settings of k, recalling that k represents the number of valid candidate test cases sampled in each iteration of the sampling phase, as described in Section 3.4. According to Table 4, when k is set to a larger value, *SamplingCA* can generate a PCA of smaller size, but costs more running time. This empirical analysis indicates that *SamplingCA* is a flexible PCAG algorithm, and can strike a good balance between effectiveness and efficiency by controlling k. As described in Section 4.4, the default setting of k in *SamplingCA* is 100. From Table 4, *SamplingCA* can achieve a good trade-off between effectiveness and efficiency using k=100.

Table 3: Average size and average time of *SamplingCA* and its alternative versions over the entire instance set except two instances (*i.e.,* `embtoolkit` and `uClinux-config`).

	SamplingCA	*Alt-1*	*Alt-2*	*Alt-3*
average size	**140.16**	556.48	188.11	922.95
average time	**34.93**	144.71	48.23	77.54

Table 4: Average size and average time of *SamplingCA* with different settings of k over the entire instance set except two instances (*i.e.,* `embtoolkit` and `uClinux-config`).

	k=10	k=50	k=100	k=500	k=1000
average size	196.35	150.12	140.16	126.34	**122.49**
average time	**5.25**	18.93	34.93	162.83	314.77

5.5 Threats to Validity

There are two potential threats to validity of our evaluations:

Random Characteristics of Competing Approaches: Actually, all competing approaches evaluated in our experiments (including *SamplingCA*, *AutoCCAG*, *FastCA* and *TCA*) are all randomized PCAG algorithms. Hence, for each approach on each instance, performing a small number of runs cannot thoroughly justify the performance of the respective approach. To reduce this potential threat, as introduced in Section 4.4, in our experiments, we conduct 100 independent runs per instance for each approach. Also, when comparing *SamplingCA* to its competitors, we conduct significance tests and calculate effect sizes to analyze the comparative results. Hence, our experimental setup can alleviate this threat to validity.

Generality of Benchmarking Instances: To mitigate this potential threat of validity, following a recent work [4], we adopt a diverse set of 125 benchmarking instances. As described in Section 4.1, these instances have been well studied in many research works [4, 29, 37, 54, 66, 68, 69]. In addition, these instances cover wide-ranging numbers of options and constraints. Hence, these instances are representative and general, so this potential threat can be reduced.

6 RELATED WORK

Combinatorial interaction testing (CIT) has been extensively explored in the last two decades, and is one of the most important research directions in software testing. For detailed literature survey, readers can refer to the books written by Kuhn *et al.* [30] and Zhang *et al.* [86], as well as review articles [64, 75]. It is well acknowledged that pairwise testing (CIT with $t = 2$) is the most common CIT technique in practice and is able to exhibit high fault detection ability in real-world applications [11, 25, 31, 74, 83].

Practical algorithms for solving the PCAG problem can be mainly divided into three classes: constraint-encoding algorithms (*e.g.,* [3, 26, 81, 87]), greedy algorithms (*e.g.,* [7–9, 11, 33–35, 76, 78, 80]) and meta-heuristic algorithms (*e.g.,* [7, 12–14, 20, 22, 23, 27, 39, 40, 49, 58]). For constraint-encoding algorithms, they first transform given PCAG instances to the instances of other combinatorial optimization problems, and then employ existing optimization solvers to solve such transformed instances. Hnich *et al.* [26], Banbara *et al.* [3], Zhang *et al.* [87] and Yamada *et al.* [81] encode the PCAG problem into the problem of constraint programming, and use powerful constraint programming solvers to deal with the encoded problem. However, such constraint-encoding algorithms can only deal with the PCAG instances with small scale.

For greedy algorithms, there are two popular types, *i.e.,* one-test-at-a-time (OTAT) methods and in-parameter-order (IPO) methods. The first OTAT method is the influential *AETG* algorithm [11]. Since then, a number of improvements have been proposed to enhance the practical performance of *AETG* [7, 8, 76, 80]. The IPO technique was proposed by Lei and Tai [35], and then Lei *et al.* introduced a general *IPOG* framework that incorporates the IPO technique [33]. Also, there are many variants of *IPOG* in the literature (*e.g.,* [34, 78]). Although greedy algorithms can deal with PCAG instances with medium scale, their constructed PCAs are usually of large size, and are impractical in real-world application scenarios where testing a single test case is much time-consuming.

Compared to both constraint-encoding algorithms and greedy algorithms, meta-heuristic algorithms can generate PCAs of much smaller size. Particularly, meta-heuristic algorithms work as follows: they try to search for a PCA of a particular size δ; once a δ-sized PCA is found, then meta-heuristic algorithms continue to find a PCA of size smaller than δ. Garvin *et al.* designed a well-known meta-heuristic algorithm called *CASA* [20, 21], which is based on advanced simulated annealing techniques. Later, Jia *et al.* improved simulated annealing through hyper-heuristic search and dynamically adapting search strategies, resulting in an enhanced algorithm called *HHSA* [27]. Lin *et al.* presented two state-of-the-art meta-heuristic algorithms named *TCA* [40] and *FastCA* [38, 39], which adopt powerful forbidden strategies to improve their practical performance. Recently, Luo *et al.* proposed a state-of-the-art algorithm dubbed *AutoCCAG* [49], which employs effective automated algorithm optimization techniques to improve its robustness. However, it is recognized that existing PCAG algorithms (including constraint-encoding ones, greedy ones and meta-heuristic ones) suffer from the serious scalability issue [68, 79]. That is, for solving large PCAG instances, existing PCAG algorithms usually take much time to generate large PCAs. Moreover, existing PCAG algorithms even fail to generate PCAs when handling huge instances. For example, as shown in Table 1, three state-of-the-art algorithms, including *AutoCCAG*, *FastCA* and *TCA*, fail to generate PCAs for two huge instances (*i.e.,* `embtoolkit` and `uClinux-config`).

For testing an SUT with many options, compared to building PCAs, there are a number of sampling methods that generate a test suite (*i.e.,* a set of test cases) to cover valid pairwise tuples as many as possible (*e.g.,* [4, 10, 17, 41, 54, 63, 65, 66, 69]). Oh *et al.* studied uniform sampling, which samples each valid test case with equal probability [66]. Baranov *et al.* presented an adaptive weighted sampling method [4], which covers more valid pairwise tuples than uniform sampling. Luo *et al.* proposed an effective sampling method called *LS-Sampling* [54]. *LS-Sampling* is based on the paradigm of local search, which shows effectiveness in solving many hard combinatorial optimization problems, including Boolean satisfiability [43, 45–47, 52, 53], maximum satisfiability [42, 44], minimum vertex cover [48], set covering [55], resource provisioning [50, 51], incident identification [24] and container reallocation [70]. Also, *LS-Sampling* exhibits the state-of-the-art performance, and greatly outperforms previous sampling methods. However, there is a serious problem for sampling methods: the test suites generated by existing sampling methods cannot achieve full pairwise coverage (*i.e.,* they cannot cover all valid pairwise tuples). According to recent empirical studies [4, 54, 66], even if a sampling method generates a test suite of large size (*e.g.,* containing thousands of test cases), such large-sized test suite still cannot achieve full pairwise coverage. Hence, using such test suites for testing would probably fail to discover a certain number of faults, incurring ineffective and inefficient testing of highly configurable systems.

In this work, we propose *SamplingCA*, which is a novel and effective sampling-based algorithm for generating PCAs. Compared to existing PCAG algorithms that suffer from the scalability issue, when solving large PCAG instances, *SamplingCA* can generate small PCAs efficiently. In particular, *SamplingCA* can build PCAs for those two huge instances (*i.e.,* `embtoolkit` and `uClinux-config`), indicating that *SamplingCA* can address the scalability issue. Compared to existing sampling approaches that cannot achieve full coverage, *SamplingCA* is able to generate PCAs, so all valid pairwise tuples are guaranteed to be covered. As a result, *SamplingCA* could bring practical benefits when testing highly configurable systems.

7 CONCLUSION

In this work, we propose a novel and efficient sampling-based approach dubbed *SamplingCA* for solving the PCAG problem. Extensive experiments on a diverse set of 125 public instances, all of which are encoded from real-world, highly configurable systems, demonstrate that *SamplingCA* can generate much smaller PCAs than all its state-of-the-art competitors (including *AutoCCAG*, *FastCA* and *TCA*), which indicates the effectiveness of *SamplingCA*. Also, our experimental results present that *SamplingCA* runs one to two magnitude faster than all its competitors, demonstrating the efficiency of *SamplingCA*. Furthermore, our empirical evaluations confirm the effectiveness of all core techniques of *SamplingCA*.

DATA AVAILABILITY STATEMENT

The implementation of *SamplingCA*, all benchmarking instances adopted in this work and detailed comparative results (including the experimental results of *SamplingCA*, all its competitors and all its alternatives on all benchmarking instances) are publicly available at **https://github.com/chuanluocs/SamplingCA** and archived at Zenodo [56].

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant 62202025 and Grant 62122078, and in part by the Natural Science Foundation of Changsha under Grant kq2202104.

REFERENCES

[1] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective Product-line Testing using Similarity-based Product Prioritization. *Software and Systems Modeling* 18, 1 (2019), 499–521.

[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] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[7] 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.

[8] 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.

[9] 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.

[10] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2015. On Parallel Scalable Uniform SAT Witness Generation. In *Proceedings of TACAS 2015.* 304–319.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] Adnan Darwiche and Knot Pipatsrisawat. 2009. Complete Algorithms. In *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 99–130.

[17] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of ICSE 2018.* 549–559.

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

[19] Jian Gao, Ruizhi Li, and Minghao Yin. 2017. A Randomized Diversification Strategy for Solving Satisfiability Problem with Long Clauses. *Science China Information Sciences* 60, 9 (2017), 092109:1–092109:11.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. 2016. Practical Minimization of Pairwise-covering Test Configurations using Constraint Programming. *Information and Software Technology* 71 (2016), 129–146.

[26] 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.

[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] 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.

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

[31] 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.

[32] Rick Kuhn, Raghu N. Kacker, Jeff Yu Lei, and Dimitris E. Simos. 2020. Input Space Coverage Matters. *Computer* 53, 1 (2020), 37–44.

[33] 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.

[34] 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.

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

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

[37] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-based Analysis of Large Real-world Feature Models is Easy. In *Proceedings of SPLC 2015.* 91–100.

[38] Jinkun Lin, Shaowei Cai, Bing He, Yingjie Fu, Chuan Luo, and Qingwei Lin. 2021. FastCA: An Effective and Efficient Tool for Combinatorial Covering Array Generation. In *Proceedings of ICSE 2021 (Companion Volume).* 77–80.

[39] Jinkun Lin, Shaowei Cai, Chuan Luo, Qingwei Lin, and Hongyu Zhang. 2019. Towards More Efficient Meta-heuristic Algorithms for Combinatorial Test Generation. In *Proceedings of ESEC/FSE 2019.* 212–222.

[40] Jinkun Lin, Chuan Luo, Shaowei Cai, Kaile Su, Dan Hao, and Lu Zhang. 2015. TCA: An Efficient Two-Mode Meta-Heuristic Algorithm for Combinatorial Test Generation. In *Proceedings of ASE 2015.* 494–505.

[41] Roberto E. Lopez-Herrejon, Francisco Chicano, Javier Ferrer, Alexander Egyed, and Enrique Alba. 2013. Multi-objective Optimal Test Suite Computation for Software Product Line Pairwise Testing. In *Proceedings of ICSM 2013.* 404–407.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Chuan Luo, Wenqian Xing, Shaowei Cai, and Chunming Hu. 2022. NuSC: An Effective Local Search Algorithm for Solving the Set Covering Problem. *IEEE Transactions on Cybernetics* (2022).

[56] Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. 2022. Artifact for ESEC/FSE 2022 Article 'SamplingCA: Effective and Efficient Sampling-Based Pairwise Testing for Highly Configurable Software Systems'. https://doi.org/10.5281/zenodo.7036747

[57] Norbert Manthey. 2011. Coprocessor - a Standalone SAT Preprocessor. In *Proceedings of INAP/WLP 2011*. 297–304.

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

[59] 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.

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

[61] 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.

[62] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC 2001*. 530–535.

[63] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don S. Batory. 2019. Uniform Random Sampling Product Configurations of Feature Models that Have Numerical Features. In *Proceedings of SPLC 2019*. 39:1–39:13.

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

[65] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of ESEC/FSE 2017*. 61–71.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] João P. Marques Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 131–153.

[73] 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.

[74] Kuo-Chung Tai and Yu Lei. 2002. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering* 28, 1 (2002), 109–111.

[75] 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.

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] Cemal Yilmaz, Sandro Fouché, Myra B. Cohen, Adam A. Porter, Gülsen Demiröz, and Ugur Koc. 2014. Moving Forward with Combinatorial Interaction Testing. *Computer* 47, 2 (2014), 37–45.

[84] 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.

[85] Hantao Zhang and Mark E. Stickel. 1996. An Efficient Algorithm for Unit Propagation. In *Proceedings of AI-MATH 1996*. 166–169.

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

[87] 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.