LS-Sampling: An Effective Local Search Based Sampling Approach for Achieving High t-wise Coverage

Chuan Luo*
Microsoft Research
Beijing, China

Binqi Sun
Microsoft Research
Beijing, China

Bo Qiao
Microsoft Research
Beijing, China

Junjie Chen
Tianjin University
Tianjin, China

Hongyu Zhang
The University of Newcastle
Callaghan, Australia

Jinkun Lin
Institute of Software, Chinese Academy of Sciences
Beijing, China

Qingwei Lin
Microsoft Research
Beijing, China

Dongmei Zhang
Microsoft Research
Beijing, China

ABSTRACT

There has been a rapidly increasing demand for developing highly configurable software systems, which urgently calls for effective testing methods. In practice, t-wise coverage has been widely recognized as a useful metric to evaluate the quality of a test suite for testing highly configurable software systems, and achieving high t-wise coverage is important for ensuring test adequacy. However, state-of-the-art methods usually cost a fairly long time to generate large test suites for high pairwise coverage (*i.e.,* 2-wise coverage), which would lead to ineffective and inefficient testing of highly configurable software systems. In this paper, we propose a novel local search based sampling approach dubbed *LS-Sampling* for achieving high t-wise coverage. Extensive experiments on a large number of public benchmarks, which are collected from real-world, highly configurable software systems, show that *LS-Sampling* achieves higher 2-wise and 3-wise coverage than the current state of the art. *LS-Sampling* is effective, since on average it achieves the 2-wise coverage of 99.64% and the 3-wise coverage of 97.87% through generating a small test suite consisting of only 100 test cases (90% smaller than the test suites generated by its state-of-the-art competitors). Furthermore, *LS-Sampling* is efficient, since it only requires an average execution time of less than one minute to generate a test suite with high 2-wise and 3-wise coverage.

CCS CONCEPTS

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

KEYWORDS

Combinatorial Interaction Testing, Local Search, Sampling

*Chuan Luo is the corresponding author (Email addresses: chuan.luo@microsoft.com, chuanluophd@outlook.com).

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468622

ACM Reference Format:
Chuan Luo, Binqi Sun, Bo Qiao, Junjie Chen, Hongyu Zhang, Jinkun Lin, Qingwei Lin, and Dongmei Zhang. 2021. *LS-Sampling*: An Effective Local Search Based Sampling Approach for Achieving High t-wise Coverage. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468622

1 INTRODUCTION

Highly configurable software systems play critical roles in many real-world applications [7, 34, 47, 71]. Such highly configurable systems usually expose many configuration options, which can be controlled by users to customize the systems [46]. However, testing highly configurable systems is challenging, since it is impractical to test all possible configurations. In practice, the number of possible configurations increases exponentially with the rise of the number of options, but only a small subset of configurations would cause systems to fail [46]. For instance, given a configurable system containing 40 options, where each option has 2 possible values, the number of total possible configurations for that system is more than one trillion (*i.e.,* $2^{40} = 1,099,511,627,776$) in the worst case.

Combinatorial interaction testing (CIT) is a practical, dominant testing paradigm for detecting option-interaction faults in highly configurable systems [7, 46]. Given a configurable system, the primary goal of CIT is to generate a small test suite to achieve high interaction coverage [38, 67]. A t-wise option combination is an option combination of t options, and t-wise coverage, which aims to achieve high interaction coverage of all t-wise option combinations, has been broadly recognized as a useful metric to evaluate the quality of a test suite for testing configurable systems [1, 2, 7, 23, 33, 38, 67, 69]. In classical CIT, various approaches (*e.g.,* [16–19, 22, 32, 44, 56, 64]) have been proposed to build test suites with full t-wise coverage (*i.e.,* all possible valid t-wise option combinations in the entire configuration space are covered). However, in practice, achieving full t-wise coverage is difficult or even infeasible when a configurable system has a large number of configuration options [69]. Also, it is acknowledged that, for highly configurable systems, test suites with full t-wise coverage are usually large, and using such test suites would exceed the testing budget [7].

Hence, for testing highly configurable systems, compared to classical approaches that focus on achieving full t-wise coverage, an

advisable solution is to solve the t-wise coverage maximum (t-wise CovMax) problem, which aims to generate a fixed-sized test suite that maximizes t-wise coverage [7]. For configurable systems, there usually exist hard constraints (mutual dependencies and exclusiveness) among different configuration options. For the sake of the accuracy and the efficiency of the testing procedures, these hard constraints must be satisfied when generating test suites, which makes the t-wise CovMax problem more complex. Effectively solving the t-wise CovMax problem is challenging from the practical perspective. In practice, achieving high t-wise coverage is difficult when the value of t is large [7] or when the option combination space is large [69]. Also, extensive empirical studies on many systems [39–41] indicate that CIT with high 2-wise coverage can detect around 77% faults, and CIT with high 3-wise coverage is able to reveal around 95% faults. Actually, recent works focus on solving the 2-wise CovMax problem [7, 69].

In CIT, uniform sampling has been recognized as a dominant domain-agnostic method for testing highly configurable systems [7, 28, 65, 69, 75]. Previous studies show that uniform sampling exhibits its effectiveness in solving the t-wise CovMax problem [7, 69]. Uniform sampling tries to sample each valid configuration with equal probability, and has been utilized to test highly configurable systems [69]. However, uniform sampling suffers from a severe issue that it cannot well handle configurable systems with complex constraints [7]. Due to the existence of complex inter-option constraints, option combinations are not uniformly distributed among the configurations. For example, a number of option combinations can appear in millions of configurations, while some option combinations can only appear in tens [7]. This phenomenon hinders uniform sampling from achieving high t-wise coverage. An empirical study [69] shows that, for a configurable system, uniform sampling can only achieve the 2-wise coverage less than 50% after generating 10^7 test cases.

Besides uniform sampling, Baranov *et al.* proposed an adaptive weighted sampling approach called *Baital* [7], which is the current state-of-the-art approach for solving the t-wise CovMax problem. As discussed before, the t-wise CovMax problem is a typical combinatorial optimization problem, and effectively solving combinatorial optimization problems requires to directly optimize the optimization objective [31, 70]. However, *Baital* does not optimize the objective of the t-wise CovMax problem explicitly during its sampling process, which would make *Baital* ineffective. Moreover, *Baital* handles hard constraints through knowledge compilation techniques [77], but the costly process of knowledge compilation would make *Baital* inefficient. In fact, our experiments (Section 5) also confirm the ineffectiveness and the inefficiency of *Baital*. Our experiments present that *Baital* has to generate a large test suite (consisting of 1,000 test cases) to achieve the average 2-wise coverage of 97.45% and the average 3-wise coverage of 93.34%, while a relatively small test suite (consisting of 100 test cases) generated by *Baital* can only obtain the average 2-wise coverage of 92.58% and the average 3-wise coverage of 81.90%. However, the capability of generating a small test suite is important, since manually configuring and testing real-world, highly configurable systems through a large test suite can be very expensive. Our experiments also show that using *Baital* for test suite generation is time-consuming. For instance, the average execution time required by *Baital* to generate 1,000 test cases is 2,822.38 seconds. However, for those projects that adopt agile software development, rapid generation of high-quality test cases is important, which urgently calls for efficient and effective test suite generation methods.

In this work, we propose a novel approach dubbed *LS-Sampling* (*Local Search based Sampling*) for achieving high t-wise coverage. Compared to state-of-the-art approaches which do not optimize the optimization objective explicitly during its sampling process, *LS-Sampling* is designed on top of an effective local search framework, whose search process is directly guided by the optimization objective. Actually, local search has exhibited its effectiveness in solving various combinatorial optimization problems (*e.g.,* [14, 26, 27, 49, 51, 55, 84]). During the local search process, *LS-Sampling* constructs the test suite in an iterative manner: in each iteration, a candidate set of valid test cases is first sampled from the entire configuration space, and then from that candidate set a valid test case that optimizes the objective most is selected. Also, *LS-Sampling* adopts a new, dynamic mechanism for updating sampling probabilities to enhance its local search process. Furthermore, different from state-of-the-art approaches, *LS-Sampling* incorporates a novel, diversity-aware heuristic search algorithm to handle hard constraints more efficiently. In addition, before the local search process, *LS-Sampling* employs powerful formula simplification techniques to equivalently simplify hard constraints, which would make our approach more efficient and effective.

Through extensive experiments, *LS-Sampling* achieves much higher t-wise coverage than uniform sampling [69] and *Baital* [7]. Our experiments are conducted on a large number of public benchmarks collected from real-world, highly configurable systems. Our experiments demonstrate that *LS-Sampling* is effective, since on average it can achieve the 2-wise coverage of 99.64% and the 3-wise coverage of 97.87% through generating a small test suite consisting of only 100 test cases, both of which are higher than the average 2-wise and 3-wise coverage achieved by the test suites consisting of 1,000 test cases generated by state-of-the-art approaches. Also, *LS-Sampling* is efficient, since it only requires the average execution time less than one minute to generate a test suite with high 2-wise and 3-wise coverage.

We summarize our major contributions of this work as follows.

- We propose a novel approach dubbed *LS-Sampling*, which is designed on top of an effective local search framework, for achieving high t-wise coverage.
- Our *LS-Sampling* approach leverages a number of core algorithmic mechanisms, including formula simplification, dynamic mechanism for updating sampling probabilities, and diversity-aware heuristic search, to enhance its both effectiveness and efficiency.
- We perform extensive experiments, indicating that our *LS-Sampling* approach can achieve higher 2-wise and 3-wise coverage through generating much smaller test suites than current state-of-the-art approaches. Also, *LS-Sampling* requires much less execution time to achieve considerably higher 2-wise and 3-wise coverage compared to *Baital*. Furthermore, our experimental results demonstrate the effectiveness of each core algorithmic mechanism underlying our *LS-Sampling* approach.

2 PRELIMINARIES

In this section, we introduce CIT and also formally describe the t-wise CovMax problem.

2.1 Combinatorial Interaction Testing

Here we introduce necessary notations related to CIT as follows.

System Under Test (SUT): An SUT is a pair $T = (P, H)$, where P denotes a set of options and H denotes a set of hard constraints on the permissible combinations of values of the options in P. Following the recent study of t-wise coverage [7], without loss of generality, this work focuses on the scenario where each option takes binary values, so in this work each option $p_i \in P$ takes binary values $\{0, 1\}$. The general scenario, where each option takes multiple discrete values, can be transformed to the binary scenario [7], and the benchmarks adopted in this work are all collected from the general scenario. Also, the configurations of an SUT can be expressed as a Boolean formula [3, 8, 62, 63, 79].

Option Combination: Given an SUT $T = (P, H)$, the definition of the option combination (also known as tuple) is a set of pairs, *i.e.,* $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \cdots, (p_{i_t}, v_{i_t})\}$, where $p_{i_j} \in P$ and $v_{i_j} \in \{0, 1\}$, indicating that option p_{i_j} takes value v_{i_j}. Specifically, an option combination of size t is called t-wise option combination.

Test Case: Given an SUT $T = (P, H)$, a test case (also known as configuration) is an option combination that covers all options in P. That is, a test case is a complete assignment to P.

In practice, the assignment to the options of a configurable system cannot violate hard constraints. Testing configurable systems with invalid test cases (*i.e.,* violating at least one hard constraint) would waste much testing time, so it is critical to ensure that all generated test cases are valid. Given an SUT $T = (P, H)$, an option combination or a test case is *valid* if it satisfies all hard constraints in H. Also, an option combination τ is *covered* by a test case α if $\tau \subseteq \alpha$, which means that each option in τ takes the same value as the one in α.

2.2 The t-wise Coverage Maximum Problem

In this subsection we first introduce Boolean formulae and then describe the t-wise coverage maximum problem.

Given a set of n Boolean *variables i.e.,* $\{x_1, x_2, \cdots, x_n\}$, a *literal* is a Boolean variable (x_i) or its negation ($\neg x_i$), and a *clause* is a disjunction of literals. A Boolean *formula* F in conjunctive normal form (CNF) is a conjunction of m clauses, *i.e.,* $F = c_1 \vee c_2 \vee \cdots \vee c_m$. Given a formula F in CNF, $V(F)$ denotes the set of Boolean variables in F, and $C(F)$ denotes the set of clauses in F. A mapping $\beta : V(F) \rightarrow \{0, 1\}$ is called an *assignment* of F. Given an assignment β, each clause has two possible *states*: *satisfied* or *unsatisfied* – a clause is satisfied if at least one literal in that clause evaluates to true under β; otherwise, that clause is unsatisfied. A *satisfying assignment* or *solution* of F is an assignment that satisfies all clauses in F.

An SUT can be represented as a Boolean formula in CNF [3, 8, 62, 63, 79]. Given an SUT $T = (P, H)$ and its corresponding Boolean formula F in CNF, T's option set P directly corresponds to F's Boolean variable set $V(F)$, where $n = |V(F)| = |P|$; T's hard constraint set H can be encoded into a set of clauses $C(F)$ [3, 8]. Furthermore, a valid test case α of T is actually a satisfying assignment β of F. A *test suite* is a set of valid test cases.

Given a valid test case α, we use the notation $Comb(\alpha, t)$ to denote the set of all valid t-wise option combinations covered by α. Given a test suite A, we can extend the notation $Comb$ and use $Comb(A, t)$ to denote the set of all valid t-wise option combinations covered by A, *i.e.,* $Comb(A, t) = \cup_{\alpha \in A} Comb(\alpha, t)$. We note that, for a single, valid test case α, $|Comb(\alpha, t)| = \binom{|P|}{t}$; however, this does not imply $|Comb(A, t)| = |A| \times \binom{|P|}{t}$, since given two different valid test cases α_1 and α_2, $|Comb(\alpha_1, t) \cup Comb(\alpha_2, t)|$ is not necessarily equal to $|Comb(\alpha_1, t)| + |Comb(\alpha_2, t)|$. Actually, in practice there are intersections among the sets of valid t-wise option combinations covered by different test cases [7].

Given an SUT T and a test suite A, the t-wise coverage of A is calculated as the ratio between the number of valid t-wise option combinations covered by A, *i.e.,* $|Comb(A, t)|$, and the total number of all possible, valid t-wise option combinations during the entire configuration space of T. Given an SUT T and its corresponding Boolean formula F, we use $S(F)$ to denote the set of all satisfying assignments of F, which can also represent the set of all valid test cases for T. Hence, the t-wise coverage of A can be formally described as $Cov(A, t) = |Comb(A, t)|/|Comb(S(F), t)|$.

In practice, for testing an SUT, the number of test cases is proportional to the time required by the entire testing process [38, 67], so it is critical to generate relatively small test suite with high t-wise coverage. To this end, this work studies the t-wise coverage maximum (t-wise CovMax) problem, aiming to generate a test suite consisting of a fixed number of test cases such that the t-wise coverage is maximized. The t-wise CovMax problem is formally described as follows: Given an SUT T, T's corresponding Boolean formula F, size of option combination t, allowed number of valid test cases k, the t-wise CovMax problem is to find a test suite A such that:

$$A = \underset{A \subseteq S(F),\ |A|=k}{\arg\max} \ Cov(A, t) \tag{1}$$

Given an SUT T and T's corresponding Boolean formula F, maximizing $Cov(A, t)$ is equivalent to maximizing $|Comb(A, t)|$, since the denominator (*i.e.,* $|Comb(S(F), t)|$) can be regarded as a constant. Hence, the t-wise CovMax problem can be expressed as follows:

$$A = \underset{A \subseteq S(F),\ |A|=k}{\arg\max} \ |Comb(A, t)| \tag{2}$$

Actually, the t-wise CovMax problem is a challenging combinatorial optimization problem [7], which calls for a practical solution.

3 THE *LS-SAMPLING* APPROACH

This section proposes *LS-Sampling* (Local Search based Sampling), a novel approach for achieving high t-wise coverage. We first introduce the top-level design of *LS-Sampling*, which is an effective local search framework for solving the t-wise CovMax problem. Then we present the core algorithmic mechanisms underlying *LS-Sampling*.

3.1 Top-Level Design of *LS-Sampling*

As introduced before, the t-wise CovMax problem is a combinatorial optimization problem [7]. Since local search [31] has exhibited its effectiveness in a variety of combinatorial optimization problems (*e.g.,* [14, 26, 27, 49, 51, 55, 84]), we are devoted to proposing a novel local search based sampling approach for solving the t-wise CovMax problem.

Algorithm 1: Top-level Design of *LS-Sampling*

Input: F: Boolean formula;
k: allowed number of valid test cases;
Output: A: the set of generated, valid test cases;
1 $F^* \leftarrow$ Simplify F through a simplification tool;
2 $A \leftarrow \emptyset$;
3 Initialize $prob(x_i)$ as 0.5 for each Boolean variable x_i in F^*;
4 **for** $iter \leftarrow 1$ **to** k **do**
5 $\quad \alpha^* \leftarrow PickTestCase(F^*, prob, A)$;
6 $\quad A \leftarrow A \cup \{\alpha^*\}$;
7 $\quad$ **foreach** $x_i \in V(F^*)$ **do**
8 $\quad\quad prob(x_i) \leftarrow UpdateProb(x_i, A)$;
9 **return** A;

Here we present and describe the top-level design of *LS-Sampling*, a local search framework that forms the core of *LS-Sampling*. The top-level design of *LS-Sampling* is outlined in Algorithm 1, and needs two inputs: 1) the Boolean formula encoded from the target SUT, denoted by F; 2) the allowed number of valid test cases for the target SUT, denoted by k. The output of our *LS-Sampling* approach is a test suite, denoted by A.

As demonstrated in Algorithm 1, our *LS-Sampling* approach is comprised of two essential phases: initialization phase and local search phase. In the initialization phase, necessary initialization steps are performed and the given Boolean formula is equivalently simplified (Lines 1–3 in Algorithm 1). In the local search phase, *LS-Sampling* constructs the test suite A in an iterative manner until $|A| = k$ (Lines 4–8 in Algorithm 1). Once the local search phase is terminated, A is output as the generated test suite (Line 9 in Algorithm 1).

3.2 Initialization Phase

Given a Boolean formula F, the approaches for solving the t-wise CovMax problem aim to identify a set of satisfying assignments (*i.e.,* valid test cases) for F that maximizes t-wise coverage. Existing state-of-the-art approaches for solving the t-wise CovMax problem, including uniform sampling [69] and *Baital* [7], directly sample the satisfying assignments for the original formula F. However, finding satisfying assignments for a Boolean formula is challenging: in theory even seeking one satisfying assignment for a Boolean formula, which is also known as Boolean satisfiability (SAT) solving in the research community of SAT, is NP-hard due to the complex, hard constraints [9].

It is recognized that simplifying Boolean formulae can considerably reduce the number of hard constraints [10, 21, 60, 78], and extensive empirical studies (*e.g.,* [5, 10, 66, 74]) show that simplifying Boolean formulae can significantly improve the effectiveness of seeking satisfying assignments. Hence, it is advisable to equip *LS-Sampling* with formula simplification techniques.

In the context of SAT solving, the primary goal is to find only one satisfying assignment for the given formula, so the widely-used simplification tools are designed to incorporate satisfiability-preserving techniques, which do not change the satisfiability status of the original formula F but alter the set of all satisfying assignments (*i.e.,* $S(F)$). However, as discussed in Section 2.2, such satisfiability-preserving techniques are not applicable to solving the t-wise CovMax problem, which aims to find a subset of $S(F)$ that maximizes the t-wise coverage.

Hence, the core problem in the initialization phase is how to identify an effective formula simplification tool that is suitable in our scenario. For solving the t-wise CovMax problem, equivalence-preserving techniques are required, which can considerably reduce the number of constraints and meanwhile guarantee that the simplified and the original formulae are equivalent. Among existing simplification tools [10, 21, 60, 78], we adopt *Coprocessor* [60], since it is publicly available[1] and integrates a number of effective equivalence-preserving techniques, *e.g.,* hidden tautology elimination [29], probing [59], asymmetric branching [73] and extended resolution [4].

In the initialization phase, *LS-Sampling* first simplifies the given formula F, resulting in the simplified formula F^*, through the formula simplification tool *Coprocessor* with the hyper-parameter setting that only activates equivalence-preserving techniques and switches off all satisfiability-preserving techniques. After that, *LS-Sampling* initializes the set of valid test cases A as empty set, and initializes the sampling probability for each Boolean variable appearing in F^* (which is an important concept in the local search phase and will be introduced in detail in Section 3.3).

3.3 Local Search Phase

Once the initialization phase is finished, *LS-Sampling* steps into the local search phase. In the local search phase, *LS-Sampling* works in an iterative manner to construct a test suite A (Lines 4–8 in Algorithm 1): in each iteration, *LS-Sampling* first selects a valid test case α^* from the set of all satisfying assignments of the simplified formula F^* (*i.e.,* $S(F^*)$) through the *PickTestCase* component; then *LS-Sampling* adds α^* into A; at the end of each iteration, *LS-Sampling* adaptively updates each Boolean variable's sampling probability through the *UpdateProb* component, a novel dynamic mechanism for updating sampling probabilities. The iteration process would be terminated once there are k valid test cases in A.

Here we define each variable x_i's sampling probability, denoted by $prob(x_i)$, as follows.

Definition 3.1. Given a Boolean formula F, for each Boolean variable $x_i \in V(F^*)$, $prob(x_i)$ denotes the sampling probability associated with x_i, representing the probability that the value of Boolean variable x_i is sampled as 1.

That is to say, the probability that the value of Boolean variable x_i is sampled as 0 can be calculated as $1 - prob(x_i)$. We note that each variable's sampling probability would be dynamically updated during the iterative search process.[2]

From Algorithm 1, it is apparent that the *PickTestCase* component (Line 5 in Algorithm 1) and the *UpdateProb* component (Line 8 in Algorithm 1) are crucial. We will present the *PickTestCase* component as follows, and will describe the *UpdateProb* component in Section 3.3.1.

[1] https://github.com/nmanthey/riss-solver
[2] For the first iteration, all variables' sampling probabilities are initialized as 0.5, which is equivalent to uniform sampling (Line 3 in Algorithm 1).

Algorithm 2: The *PickTestCase* Component

Input: F^*: simplified Boolean formula;
$\boldsymbol{prob}$: vector of all variables' sampling probabilities;
$\boldsymbol{A}$: current set of valid test cases;
Output: $\boldsymbol{\alpha^*}$: the selected, valid test cases;

$C \leftarrow \emptyset$;
for $j \leftarrow 1$ **to** λ **do**
  $\alpha_j \leftarrow$ Generate a test case by sampling the value of each variable x_i according to $prob(x_i)$;
  $\alpha'_j \leftarrow DAHS(F^*, \alpha_j, prob)$;
  $C \leftarrow \{\alpha'_j\}$;
  Calculate $score(\alpha'_j)$ for the valid test case α'_j;
$\alpha^* \leftarrow$ the valid test case in C with the largest *score*;
return α^*;

The *PickTestCase* component works in a greedy manner, and the primary goal of the *PickTestCase* component is to select a valid test case α^*, whose addition into A maximizes the t-wise coverage of $A \cup \{\alpha^*\}$, from $S(F^*)$. However, there are two challenges in this component,*i.e.,* metric challenge and scale challenge, as described as follows:

- **Metric Challenge:** Given a set of different valid test cases C, it is critical to design a metric that can effectively quantify the benefit of each valid test case $\alpha \in C$ with regard to A if α is added into A.
- **Scale Challenge:** In practice the cardinality of $S(F^*)$ would be extremely large, and $|S(F^*)|$ grows exponentially with the rise of the number of variables in F^*. Therefore, it is infeasible to traverse the entire set $S(F^*)$, and select one satisfying assignment from $S(F^*)$ due to the scale challenge.

In order to address the metric challenge, we design a scoring function *score* to reflect the benefit of each valid test case over A.

Definition 3.2. Given a valid test case α and a test suite A, $score(\alpha)$ denotes the increment of the number of covered t-wise option combinations if α is added into A, *i.e.,* $score(\alpha) = |Comb(A, t) \cup Comb(\alpha, t)| - |Comb(A, t)|$.

According to Definition 3.2, our scoring function can directly quantify each valid test case's unique contribution to the optimization objective of the t-wise CovMax problem (Equation 2). Given a valid test case α, the computational complexity for calculating $Comb(\alpha, t)$ is $O(\binom{n}{t})$, where n represents the number of variables. It is obvious that calculating $score(\alpha)$ with large value of t is time consuming. Therefore, it is important to propose an alternative scoring function that can be calculated efficiently.

It is recognized that, given a test suite A, if A could achieve high 2-wise coverage, then A would also obtain high t-wise coverage with $t \geqslant 3$ [7], and the empirical evidences can be witnessed in an empirical study [71] and our experiments (in Section 5). Thus, in *LS-Sampling*, $score(\alpha)$ is implemented as $score(\alpha) = |Comb(A, 2) \cup Comb(\alpha, 2)| - |Comb(A, 2)|$, so its computational complexity is just $O(n^2)$, which in turn makes the process of calculating *score* much more efficient. We note that, through such efficient implementation of *score*, the test suite generation process of *LS-Sampling* is independent of the value of t. Thus, given an SUT, the final test suite generated by *LS-Sampling* remains the same for solving the t-wise CovMax problem on that SUT even with different values of t.[3]

Algorithm 3: The *UpdateProb* Component

Input: $\boldsymbol{x_i}$: Boolean variable;
$\boldsymbol{A}$: current set of valid test cases;
Output: $\boldsymbol{prob(x_i)}$: updated sampling probability for x_i;

$B \leftarrow$ the set of all x_i-false test cases in A;
$prob(x_i) \leftarrow |B|/|A|$;
return $prob(x_i)$;

Here we switch to introduce how to address the scale challenge. *LS-Sampling* first tries to sample a candidate set of λ valid test cases $C \subset S(F^*)$, and then select a valid test case with the largest *score* from C. Through this way, in each iteration *LS-Sampling* only needs to traverse the candidate set C consisting of λ valid test cases, rather than traversing the entire set $S(F^*)$. Hence, this practical method effectively alleviates the scale challenge. We note that λ is a hyper-parameter of *LS-Sampling*, which controls the trade-off between the effectiveness and the efficiency of *LS-Sampling* and makes our *LS-Sampling* approach flexible: if λ is set to a relatively large integer, then *LS-Sampling* would achieve higher t-wise coverage but requires longer execution time; otherwise, *LS-Sampling* would run faster with a relatively lower t-wise coverage. The effect of hyper-parameter λ will be analyzed in Section 5.5.

The *PickTestCase* component is outlined in Algorithm 2. The *PickTestCase* component generates the candidate set C consisting of λ valid test cases one by one. When generating j-th valid test case, *PickTestCase* first generates a test case α_j by sampling the value of each Boolean variable x_i according to its sampling probability $prob(x_i)$. Recalling that there are hard constraints with regard to formula F^*, since α_j is sampled without considering such hard constraints, α_j would possibly be invalid. To address this issue, we propose *Diversity-aware Heuristic Search* (*DAHS*), a novel algorithm that conducts heuristic search to modify α_j, resulting in a valid test case α'_j (our *DAHS* algorithm will be presented in Section 3.3.2). Once the candidate set C is constructed, the *PickTestCase* component returns the valid test case $\alpha^* \in C$ with the largest *score*.

Different from *Baital*, the local search process of *LS-Sampling* is explicitly guided by the optimization objective of the t-wise CovMax problem, since our proposed metric *score*, utilized in the *PickTestCase* component, can reflect each valid test case's unique contribution to the optimization objective.

3.3.1 Dynamic Mechanism for Updating Sampling Probabilities. As presented in Section 3.3, the value of each variable's sampling probability directly impacts the quality of the valid test case selected by *PickTestCase*. Actually, *PickTestCase* aims to select a valid test case α^*, whose addition into A maximizes the t-wise coverage of $A \cup \{\alpha^*\}$. Hence, the test cases in A needs to be diversified, so there is a requirement of diversification when generating test cases.

For solving the t-wise CovMax problem, uniform sampling, as a dominant domain-agnostic method, adopts a simple, static mechanism to decide the values of those sampling probabilities (*i.e.,* setting

[3] Actually, *LS-Sampling* is a randomized approach. This conclusion holds when the random seed for *LS-Sampling* is fixed.

Algorithm 4: The *DAHS* Algorithm

Input: F^*: simplified Boolean formula;
$\boldsymbol{\alpha}$: the input, (possibly invalid) test case;
$\boldsymbol{prob}$: vector of all variables' sampling probabilities;
Output: $\boldsymbol{\alpha'}$: the modified, valid test case;

```
α' ← α;
while α' is invalid do
    if DV ≠ ∅ then
        x* ← the variable with the largest gain in DV, breaking
          ties by preferring the variable with the largest priority;
    else
        c ← a random unsatisfied clause;
        x* ← the variable with the largest priority in c;
    α' ← α' with x* flipped;
return α';
```

each Boolean variable's sampling probability to 0.5), such that the value of each Boolean variable is sampled uniformly at random. However, the static mechanism utilized by uniform sampling keeps all the sampling probabilities as fixed values during the entire process of test suite generation, so it neither considers the current status of A nor reflects the requirement of diversification when generating test cases.

To meet the requirement of diversification, an advisable method is to dynamically update each Boolean variable's sampling probability according to the test cases in A so far. To this end, we propose a novel dynamic mechanism for updating sampling probabilities. Given a Boolean variable x_i, a test case α is defined as x_i-false test case if x_i's value under α is 0, and a test case α is defined as x_i-true test case if x_i's value under α is 1. According to Definition 3.1, given a variable x_i, $prob(x_i)$ is the probability that x_i's value is sampled as 1. Considering the current status of A, it is advisable to update x_i's sampling probability as the ratio between the number of x_i-false test cases in A and the number of all test cases in A. The procedures of our proposed dynamic mechanism for updating sampling probabilities are outlined in Algorithm 3.

3.3.2 Diversity-Aware Heuristic Search. As discussed in Section 3.3, directly sampling test cases according to sampling probabilities *prob* would possibly result in invalid test cases, due to hard constraints with regard to F^*. Hence, given an invalid test case α, it is important to design an effective method to modify α and convert it to a valid test case α'. Actually, heuristic search based SAT algorithms (*e.g.*, [35, 36, 50, 52–54, 57, 58]) repeatedly modifies the input assignment through flipping the truth value of a single Boolean variable, until a satisfying assignment is reached. Since a test case is an assignment regarding the input formula F^* (as previously discussed in Section 2), thus in our scenario a simple idea is to adopt an existing heuristic search based SAT algorithm in *LS-Sampling* to process the possibly invalid test case α.

However, existing heuristic search based SAT algorithms focus on finding a satisfying assignment fast, but does not consider the requirement of diversification during its assignment modification process. As discussed in Section 3.3.1, ignoring the requirement of diversification would make those test cases in A less diversified and thus hinders the t-wise coverage of A from being maximized, so in our opinion this is a severe issue when directly applying existing heuristic search based SAT algorithms in our scenario.

In order to address this issue, we propose a novel diversity-aware heuristic search (*DAHS*) algorithm. As discussed in Section 3.3.1, sampling probabilities can effectively reflect the requirement of diversification. Hence, different from existing heuristic search based SAT algorithms, our *DAHS* algorithm innovatively makes the use of sampling probabilities during its modification process, and thus is capable of being aware of diversity.

Before presenting our *DAHS* algorithm, we formally define two crucial metrics to select flipping variables, *i.e.*, *gain* and *priority*.

Definition 3.3. Given a Boolean variable x_i, the *gain* of x_i, denoted by $gain(x_i)$, is the decrement in the number of unsatisfied clauses by flipping the current truth value of x_i.

We note that a variable's *gain* could be negative, since it is possible that flipping a variable might result in more unsatisfied clauses. For a variable x_i, x_i is a *decreasing variable* if $gain(x_i) > 0$. Also, we use notation DV to denote the current set of all decreasing variables. *DAHS* prefers to flip decreasing variables, since flipping decreasing variables would reduce more unsatisfied clauses.

Definition 3.4. Given a Boolean variable x_i and x_i's current truth value v_i, the *priority* of x_i, denoted by $priority(x_i)$, is calculated as $(1 - v_i) \times prob(x_i) + v_i \times (1 - prob(x_i))$.

According to Definition 3.4, if variable x_i's current truth value v_i is 0, then $priority(x_i)$ becomes $prob(x_i)$; otherwise (if v_i is 1), $priority(x_i)$ is calculated as $1 - prob(x_i)$. Our *priority* metric is designed based on sampling probability, and thus utilizing *priority* to select flipping variables can make *DAHS* be aware of diversity. To this end, *DAHS* prefers to flip such variables with large *priority*.

Our *DAHS* algorithm is presented in Algorithm 4. Given a (possibly invalid) test case, *DAHS* repeatedly selects a variable x^* and flips the truth value of x^* until a valid test case is reached. For *DAHS*, when selecting a flipping variable x^*, if the set of decreasing variables DV is not empty, then the variable with the largest *gain* in DV is selected as the flipping variable x^*, breaking ties by preferring the variable with the largest *priority*; otherwise, the variable with the largest *priority* in a random unsatisfied clause is selected as the flipping variable x^*. Through this way, *DAHS* can return a valid test case and meanwhile can better reflect the requirement of diversification than existing heuristic search based SAT algorithms.

4 EXPERIMENTAL DESIGN

In this section, we describe the experimental design of this work. Particularly, we first introduce the benchmarks adopted in our experiments. Then we describe the state-of-the-art competitors of *LS-Sampling*. After that, we present the research questions of this work. Finally, we describe the experimental setup.

4.1 Benchmarks

In order to study the practical performance of *LS-Sampling*, in our experiments we adopt a benchmark set consisting of 123 public benchmarks. All benchmarks adopted in our experiments are originally collected by Baranov *et al.* [7]. Also, each benchmark is collected from a real-world, configurable system, and is expressed as a Boolean formula in CNF. Those benchmarks have been well

Table 1: Results of 2-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on 20 representative benchmarks.

Benchmark	#TotalOC	*Uniform* (k=1,000)		*Baital* (k=100)		*Baital* (k=1,000)		*LS-Sampling* (k=100)		*LS-Sampling* (k=1,000)	
		#OC	Cov (%)	#OC	Cov (%)	#OC	Cov (%)	#OC	Cov (%)	#OC	Cov (%)
busybox_1_28_0	1,965,023	1,903,285.00	98.69	1,949,742.00	99.22	1,954,917.15	99.49	**1,962,484.05**	**99.87**	**1,963,429.05**	**99.92**
csb281	2,873,486	2,093,606.95	77.77	2,651,087.05	92.26	2,790,106.35	97.10	**2,868,096.70**	**99.81**	**2,873,256.85**	**99.99**
dreamcast	2,908,040	2,116,793.15	78.68	2,677,222.90	92.06	2,849,932.60	98.00	**2,902,749.90**	**99.82**	**2,907,875.75**	**99.99**
ebsa285	2,928,811	2,143,454.85	78.01	2,702,292.00	92.27	2,843,430.50	97.08	**2,923,255.35**	**99.81**	**2,928,608.75**	**99.99**
ecos-icse11	2,910,229	2,141,688.70	79.43	2,693,665.75	92.56	2,849,442.55	97.91	**2,905,417.30**	**99.83**	**2,910,024.90**	**99.99**
financial	917,150	425,312.90	47.67	547,853.75	59.73	642,027.55	70.00	**712,098.30**	**77.64**	**909,548.80**	**99.17**
freebsd-icse11	3,765,597	3,345,974.50	94.21	3,703,096.50	98.34	3,720,618.90	98.81	**3,734,605.40**	**99.18**	**3,746,891.50**	**99.50**
hs7729pci	3,150,211	2,263,428.15	76.32	2,895,653.50	91.92	3,054,298.15	96.96	**3,141,987.45**	**99.74**	**3,149,968.50**	**99.99**
integrator_arm9	2,998,857	2,175,671.00	77.80	2,776,654.15	92.59	2,928,102.10	97.64	**2,992,863.65**	**99.80**	**2,998,648.15**	**99.99**
linux	2,797,796	2,222,435.35	79.44	2,593,698.90	92.71	2,726,210.40	97.44	**2,792,832.00**	**99.82**	**2,797,612.65**	**99.99**
mpc50	2,719,748	1,994,474.75	78.85	2,525,240.40	92.85	2,664,419.45	97.97	**2,715,982.80**	**99.86**	**2,719,589.90**	**99.99**
ocelot	2,986,129	2,178,193.10	78.36	2,751,625.40	92.15	2,911,887.90	97.51	**2,981,106.75**	**99.83**	**2,985,886.40**	**99.99**
olpce2294	3,037,775	2,204,305.15	78.08	2,799,294.10	92.15	2,951,585.20	97.16	**3,032,389.40**	**99.82**	**3,037,596.25**	**99.99**
olpcl2294	3,033,027	2,195,703.75	77.80	2,791,027.65	92.02	2,949,549.75	97.25	**3,027,799.15**	**99.83**	**3,032,830.85**	**99.99**
pati	2,901,007	2,062,894.90	76.69	2,665,798.60	91.89	2,815,691.15	97.06	**2,896,516.40**	**99.85**	**2,900,847.25**	**99.99**
pc_i82544	2,977,432	2,180,289.15	78.72	2,756,109.95	92.57	2,904,523.75	97.55	**2,971,954.45**	**99.82**	**2,977,215.70**	**99.99**
phycore	3,008,140	2,224,954.85	78.63	2,786,701.15	92.64	2,904,478.70	96.55	**3,003,326.80**	**99.84**	**3,007,907.65**	**99.99**
refidt334	3,022,264	2,219,398.25	78.29	2,792,108.55	92.38	2,944,856.95	97.44	**3,016,248.75**	**99.80**	**3,022,026.05**	**99.99**
vrc4373	2,884,611	2,132,122.50	78.93	2,675,113.10	92.74	2,780,506.00	96.39	**2,880,095.45**	**99.84**	**2,884,388.50**	**99.99**
XSEngine	2,974,825	2,169,775.25	78.49	2,752,925.45	92.54	2,896,382.30	97.36	**2,969,901.90**	**99.83**	**2,974,596.80**	**99.99**

studied in literature [7, 37, 45, 69, 72, 75]. For all benchmarks in our benchmark set, the number of Boolean variables ranges from 94 to 2,128, and the number of clauses varies from 190 to 62,183. All benchmarks used in our experiments are available online.[4]

4.2 Competitors

In our experiments, *LS-Sampling* is compared against two state-of-the-art competitors, *i.e.,* uniform sampling [69] and *Baital* [7].

Uniform sampling has been recognized as a dominant domain-agnostic method to achieve high t-wise coverage, and it is capable of generating test suites for highly configurable systems [69]. In our experiments, we adopt the same implementation of uniform sampling as the one evaluated in the literature [7].

Baital is a recently-proposed, adaptive weighted sampling approach, which is the current state-of-the-art approach for solving the t-wise CovMax problem [7]. As reported in the literature [7], *Baital* achieves higher 2-wise coverage than uniform sampling.

We note that the implementations of both uniform sampling and *Baital* are available online.[5]

4.3 Research Questions

To evaluate the effectiveness of *LS-Sampling*, we aim to answer the following research questions (RQs). In practice, t-wise coverage has been widely recognized as a critical metric to evaluate the quality of a test suite for testing highly configurable systems [1, 2, 7, 23, 33, 38, 67, 69]. Since achieving high t-wise coverage can be difficult for large values of t [7], the practitioners mainly generate test suites with high 2-wise or 3-wise coverage when testing configurable systems [7]. Hence, in our experiments we focus on advancing the current state of the art in achieving high 2-wise coverage and high 3-wise coverage.

RQ1: Can *LS-Sampling* achieve higher 2-wise coverage than its state-of-the-art competitors (*i.e.,* uniform sampling and *Baital*)?

In this RQ, *LS-Sampling* is compared against two state-of-the-art competitors, *i.e.,* uniform sampling [69] and *Baital* [7], for solving the 2-wise CovMax problem.

RQ2: Can *LS-Sampling* achieve higher 3-wise coverage than its state-of-the-art competitors (*i.e.,* uniform sampling and *Baital*)?

Empirical studies on many real-world, configurable systems [39–41] present that test suites with high 2-wise coverage and with high 3-wise coverage can detect around 77% faults and 95% faults, respectively, indicating that achieving high 3-wise coverage is important and is complementary with achieving high 2-wise coverage. In this RQ, *LS-Sampling* is compared against uniform sampling [69] and *Baital* [7] for solving the 3-wise CovMax problem.

RQ3: How efficient is *LS-Sampling* to generate a test suite with high 2-wise and 3-wise coverage?

In this RQ, we evaluate the execution time of *LS-Sampling* to generate a test suite with high 2-wise and 3-wise coverage.

RQ4: How effective is each core algorithmic mechanism underlying *LS-Sampling*?

In this RQ, we study the effectiveness of *LS-Sampling*'s core algorithmic mechanisms, including formula simplification (Section 3.2), dynamic mechanism for updating sampling probabilities (Section 3.3.1) and diversity-aware heuristic search (Section 3.3.2).

RQ5: How does the hyper-parameter setting affect the performance of *LS-Sampling*?

In this RQ, we analyze how the setting of hyper-parameter λ in Algorithm 2 affects the performance of *LS-Sampling*.

4.4 Experimental Setup

All experiments in this work were conducted on a workstation equipped with Intel Xeon E5-2673 CPU and 256 GB memory, running GNU/Linux. Since *LS-Sampling* and its two competitors (*i.e.,* uniform sampling and *Baital*) are all randomized approaches, we

[4] https://github.com/chuanluocs/LS-Sampling
[5] https://github.com/meelgroup/baital

Table 2: Results of 3-wise coverage achieved by *LS-Sampling* and *Baital* on 20 representative benchmarks.

Benchmark	#TotalOC	*Baital* (k=100)		*Baital* (k=1,000)		*LS-Sampling* (k=100)		*LS-Sampling* (k=1,000)	
		#OC	Cov (%)	#OC	Cov (%)	#OC	Cov (%)	#OC	Cov (%)
busybox_1_28_0	1,295,475,693	1,273,248,109.55	98.28	1,284,516,632.75	99.15	**1,290,162,984.20**	**99.59**	**1,293,707,897.90**	**99.86**
csb281	2,204,475,868	1,779,164,490.50	80.71	2,028,556,523.50	92.02	**2,158,355,263.30**	**97.91**	**2,201,507,923.75**	**99.87**
dreamcast	2,246,891,653	1,805,656,104.05	80.36	2,126,105,777.30	94.62	**2,201,328,111.70**	**97.97**	**2,244,069,470.90**	**99.87**
ebsa285	2,269,216,797	1,831,148,801.85	80.70	2,088,928,606.35	92.06	**2,221,109,726.65**	**97.88**	**2,266,093,377.80**	**99.86**
ecos-icse11	2,250,481,834	1,827,314,844.50	81.20	2,116,980,096.75	94.07	**2,206,533,610.80**	**98.05**	**2,247,615,118.85**	**99.87**
financial	332,165,284	175,455,841.25	52.82	218,003,249.25	65.63	**242,243,897.40**	**72.93**	**325,637,307.20**	**98.03**
freebsd-icse11	3,420,415,512	3,294,770,333.65	96.33	3,346,967,546.70	97.85	**3,319,425,139.65**	**97.05**	**3,389,883,227.70**	**99.11**
hs7729pci	2,532,796,273	2,026,953,758.60	80.03	2,319,863,886.60	91.59	**2,470,185,973.15**	**97.53**	**2,528,743,895.15**	**99.84**
integrator_arm9	2,353,633,751	1,918,093,759.65	81.49	2,193,392,472.20	93.19	**2,302,260,345.45**	**97.82**	**2,350,335,261.85**	**99.86**
linux	2,119,674,243	1,731,278,897.80	81.68	1,969,108,398.55	92.90	**2,076,887,256.25**	**97.98**	**2,116,958,349.55**	**99.87**
mpc50	2,033,280,572	1,668,229,762.90	82.05	1,913,209,176.10	94.09	**1,997,735,137.50**	**98.25**	**2,031,073,284.85**	**99.89**
ocelot	2,338,414,562	1,892,233,022.75	80.92	2,175,644,611.75	93.04	**2,293,530,356.30**	**98.08**	**2,335,455,040.70**	**99.87**
olpce2294	2,399,140,512	1,928,051,249.95	80.36	2,210,818,948.00	92.15	**2,350,296,073.95**	**97.96**	**2,396,085,980.30**	**99.87**
olpcl2294	2,393,625,083	1,919,000,163.20	80.17	2,207,964,220.50	92.24	**2,345,283,551.75**	**97.98**	**2,390,520,606.60**	**99.87**
pati	2,238,542,159	1,792,426,104.90	80.07	2,055,344,663.20	91.82	**2,196,671,078.90**	**98.13**	**2,235,981,891.60**	**99.89**
pc_i82544	2,328,925,905	1,891,810,484.55	81.23	2,165,283,637.30	92.97	**2,281,128,643.50**	**97.95**	**2,325,798,151.35**	**99.87**
phycore	2,366,208,638	1,938,457,404.60	81.92	2,166,307,257.85	91.55	**2,321,830,914.85**	**98.12**	**2,363,307,053.45**	**99.88**
refidt334	2,379,424,764	1,933,507,540.50	81.26	2,213,310,380.85	93.02	**2,328,915,115.85**	**97.88**	**2,376,111,853.55**	**99.86**
vrc4373	2,221,226,554	1,822,837,244.80	82.06	2,031,956,311.55	91.48	**2,180,868,934.05**	**98.18**	**2,218,509,381.95**	**99.88**
XSEngine	2,327,972,654	1,899,748,518.15	81.61	2,168,790,835.60	93.16	**2,282,763,620.85**	**98.06**	**2,325,112,888.95**	**99.88**

performed 20 independent runs per benchmark for each approach. For solving the t-wise CovMax problem with $t = 2$ and $t = 3$, the allowed number of valid test cases k is set to 1,000, following the experimental setup of the most recent study [7]. Besides, we also conduct experiments to compare *LS-Sampling* against its competitors on solving the 2-wise and 3-wise CovMax problems with k=100, to show that *LS-Sampling* can achieve both high 2-wise and 3-wise coverage through generating much smaller test suites.

In our experiments, for *LS-Sampling*, λ is set to 100. We note that the impact of different hyper-parameter settings of λ is studied in Section 5.5. For uniform sampling, we adopt its implementation as the same one used in the literature [7]. For *Baital*, it is evaluated using the default configuration recommended by its authors [7].

When solving the t-wise CovMax problem with t=2 and t=3, for each approach on each benchmark, we report the average number of t-wise option combinations covered by the generated test suite over 20 runs, denoted by '#OC', as well as the average t-wise coverage achieved by the generated test suite over 20 runs, denoted by 'Cov' (in Tables 1 and 2). Furthermore, we also summarize the average t-wise coverage over all benchmarks (in Tables 3, 5 and 7). In our experiments, for each benchmark or benchmark set, we use **boldface** to indicate the best results with regard to '#OC' and 'Cov' within the comparisons. Besides, for each benchmark, we also list the total number of possible valid t-wise option combinations in the entire configuration space, denoted by '#TotalOC', for reference. To save space, in Tables 1–4 and Figure 1, we use '*Uniform*' to denote uniform sampling.

In addition, to demonstrate the efficiency of *LS-Sampling*, we present the average execution time over all benchmarks, denoted by 'time' (in Tables 4, 6 and 8). As discussed in Section 3.3, given a benchmark, the final test suite generated by *LS-Sampling* (with the fixed random seed) is the same for solving the t-wise CovMax problem regardless of the value of t. Similarly, according to the literature [7], uniform sampling and *Baital* (with fixed random seeds) would also generate the same test suite for solving the t-wise CovMax problem on the given benchmark regardless of the values of t. Hence, on the same benchmark set, the average execution time required by each of *LS-Sampling*, uniform sampling, and *Baital* for solving the 2-wise CovMax problem is the same as that for solving the 3-wise CovMax problem, respectively.

Moreover, in Tables 1 and 2, regarding the metrics of '#OC' and 'Cov', for each benchmark, we individually compare the performance of *LS-Sampling* against that of each competitor; we conduct Wilcoxon signed-rank tests to check the statistical significance of the results and calculate the Vargha-Delaney effect sizes [82] for each pairwise comparison between *LS-Sampling* and each of its competitors. For each benchmark, if 1) all the p-values of Wilcoxon signed-rank tests at 95% confidence level are smaller than 0.05, and 2) the Vargha-Delaney effect sizes for all pairwise comparisons are larger than 0.71 (indicating large effect sizes) [76, 82], we consider the performance improvement of *LS-Sampling* over all its competitors statistically significant and meaningful, and mark these results with underline. In addition, in Tables 3 and 5, for each benchmark set, we conduct the similar process to examine whether the improvement of *LS-Sampling* on t-wise coverage over all its competitors statistically significant and meaningful, and mark the results of *LS-Sampling* with underline if this is the case.

5 EXPERIMENTAL RESULTS

In this section, we first report the experimental results, and then we discuss the threats to validity.

5.1 RQ1: Comparison on 2-wise Coverage

The results of the 2-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on 20 selected benchmarks are demonstrated in Table 1. For those 20 selected benchmarks, 10 of them are identified as representative ones by Baranov *et al.* [7], and the other 10 benchmarks are selected randomly. To save space, Table 1 does not report

Table 3: Average 2-wise coverage and average 3-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on all benchmarks.

	Uniform Cov (%)	*Baital* Cov (%)	*LS-Sampling* Cov (%)
2-wise (k=100)	73.70	92.58	**99.64**
2-wise (k=1,000)	78.90	97.45	**99.98**
3-wise (k=100)	63.19	81.90	**97.87**
3-wise (k=1,000)	70.75	93.34	**99.85**

Table 4: Average execution time required by *LS-Sampling*, uniform sampling and *Baital* for test suite generation on all benchmarks.

	Uniform	*Baital*	*LS-Sampling*
time (k=100)	275.98 sec	1,705.63 sec	39.00 sec
time (k=1,000)	322.27 sec	2,822.38 sec	400.64 sec

the 2-wise coverage achieved by uniform sampling with k=100 on those representative benchmarks. Actually, the 2-wise coverage achieved by uniform sampling with k = 100 is considerably lower than that achieved by other competing approaches. The full results of the 2-wise coverage achieved by all competing approaches with both k=100 and k=1,000 on all benchmarks are available online.[4] Also, Table 3 summarizes the average 2-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on all benchmarks.

From Tables 1 and 3, for both scenarios under k=100 and k=1,000, *LS-Sampling* achieves considerably higher 2-wise coverage than uniform sampling and *Baital*. Also, from Table 3, *LS-Sampling* with k=100 achieves the average 2-wise coverage of 99.64% on all benchmarks, while the existing state-of-the-art approach *Baital* with k=100 and k=1,000 obtains the average 2-wise coverage of 92.58% and 97.45%, respectively, indicating that our *LS-Sampling* approach achieves higher 2-wise coverage through generating much smaller test suites than the current state of the art.

5.2 RQ2: Comparison on 3-wise Coverage

Table 2 reports the results of the 3-wise coverage achieved by *LS-Sampling* and *Baital* on 20 representative benchmarks. To save space, Table 2 does not report the 3-wise coverage achieved by uniform sampling on those representative benchmarks. In fact, uniform sampling obtains much lower 3-wise coverage than *LS-Sampling* and *Baital*. The full results of the 3-wise coverage achieved by all competing approaches (including uniform sampling) on all benchmarks are available online[4]. Also, the average 3-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on all benchmarks are presented in Table 3.

From Tables 2 and 3, for both scenarios under k=100 and k=1,000, the 3-wise coverage achieved by *LS-Sampling* is much higher than that achieved by uniform sampling and *Baital*. More encouragingly, from Table 2, *LS-Sampling* with k=100 achieves the average 3-wise coverage of 97.87% on all benchmarks, while that figure for *Baital* with k=1,000 is 93.34%. Also, the average 3-wise coverage achieved by *Baital* with k=100 is only 81.90%, which is significantly worse than that achieved by *LS-Sampling* with k=100. The results clearly demonstrates that *LS-Sampling* obtains considerably higher 3-wise coverage through generating much smaller test suites than the current state of the art.

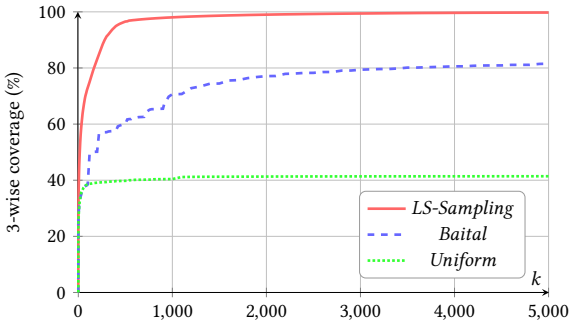

Figure 1: 3-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* on benchmark `financial`.

Furthermore, we illustrate the 3-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* with k up to 5,000 for the `financial` benchmark in Figure 1. The `financial` benchmark is particularly interesting and has been well studied in previous studies [7, 69], since both uniform sampling and *Baital* do not achieve high 2-wise and 3-wise coverage even with large test suites [7, 69]. According to Figure 1, *LS-Sampling* achieves consistently higher 3-wise coverage than uniform sampling and *Baital* in various scenarios under different values of k, confirming the effectiveness of *LS-Sampling* for achieving high 3-wise coverage.

5.3 RQ3: Comparison on Efficiency

To study the efficiency of *LS-Sampling*, we present the average execution time required by *LS-Sampling*, uniform sampling and *Baital* for test suite generation on all benchmarks in Table 4. For the scenario under k=100, *LS-Sampling* requires the average time of 39.00 seconds, which is faster than uniform sampling (275.98 seconds) and *Baital* (1,705.63 seconds). For the scenario under k=1,000, the average execution time required by *LS-Sampling*, uniform sampling and *Baital* is 400.64, 322.27 and 2,822.38 seconds, respectively. The results in Tables 3 and 4 indicate that *LS-Sampling* with k=100 is both efficient and effective, since it requires much less execution time but achieves considerably higher 2-wise and 3-wise coverage compared to *Baital*.

5.4 RQ4: Effects of Algorithmic Mechanisms

There are 3 core algorithmic mechanisms underlying *LS-Sampling*, including formula simplification (Section 3.2), dynamic mechanism for updating sampling probabilities (Section 3.3.1), and diversity-aware heuristic search (Section 3.3.2). To evaluate the effectiveness of each algorithmic mechanism, we modify *LS-Sampling* by removing each of formula simplification, dynamic mechanism for updating sampling probabilities, and diversity-aware heuristic search, resulting in three alternative versions of *LS-Sampling*, dubbed *LS-Sampling-alt1*, *LS-Sampling-alt2*, *LS-Sampling-alt3*, respectively.

Table 5 summarizes the average 2-wise coverage and the average 3-wise coverage achieved by *LS-Sampling* and its three alternative versions on all benchmarks. Table 6 reports the average execution time required by *LS-Sampling* and its three alternative versions

Table 5: Average 2-wise coverage and average 3-wise coverage achieved by *LS-Sampling* and its three alternative versions on all benchmarks.

	alt1 Cov (%)	*alt2* Cov (%)	*alt3* Cov (%)	*LS-Sampling* Cov (%)
2-wise (k=100)	99.18	97.87	99.02	**99.64**
2-wise (k=1,000)	99.85	98.82	99.79	**99.98**
3-wise (k=100)	96.54	93.79	96.56	**97.87**
3-wise (k=1,000)	99.29	97.37	99.23	**99.85**

Table 6: Average execution time required by *LS-Sampling* and its three alternative versions for test suite generation on all benchmarks.

	alt1	*alt2*	*alt3*	*LS-Sampling*
time (k=100)	57.02 sec	40.82 sec	39.48 sec	39.00 sec
time (k=1,000)	581.89 sec	411.50 sec	400.83 sec	400.64 sec

Table 7: Average 2-wise coverage and average 3-wise coverage achieved by *LS-Sampling* with different hyper-parameter settings of λ on all benchmarks.

	λ=10 Cov (%)	λ=50 Cov (%)	λ=100 Cov (%)	λ=500 Cov (%)	λ=1,000 Cov (%)
2-wise (k=100)	99.44	99.59	99.64	99.71	**99.73**
2-wise (k=1,000)	99.94	99.97	**99.98**	**99.98**	**99.98**
3-wise (k=100)	97.22	97.72	97.87	98.12	**98.20**
3-wise (k=1,000)	99.74	99.83	99.85	**99.87**	**99.87**

Table 8: Average execution time required by *LS-Sampling* with different hyper-parameter settings of λ for test suite generation on all benchmarks.

	λ=10	λ=50	λ=100	λ=500	λ=1,000
time (k=100)	4.13 sec	19.62 sec	39.00 sec	193.84 sec	387.28 sec
time (k=1,000)	42.03 sec	201.25 sec	400.64 sec	1993.06 sec	3980.29 sec

on all benchmarks. To save space, in Tables 5 and 6, we use '*alt1*', '*alt2*' and '*alt3*' to represent '*LS-Sampling-alt1*', '*LS-Sampling-alt2*' and '*LS-Sampling-alt3*', respectively. According to Tables 5 and 6, *LS-Sampling* is able to achieve both higher 2-wise and 3-wise coverage than all its alternative versions. Also, *LS-Sampling* runs faster than all its alternative versions. Hence, the results in Tables 5 and 6 indicate the effectiveness of each core algorithmic mechanism underlying *LS-Sampling*. Furthermore, from Tables 3 and 5, each alternative version of *LS-Sampling* achieves both higher 2-wise and 3-wise coverage through generating much smaller test suites than uniform sampling and *Baital*, indicating the effectiveness of the local search framework of *LS-Sampling*.

5.5 RQ5: Effect of Hyper-Parameter Setting

Table 7 reports the average 2-wise coverage and the average 3-wise coverage achieved by *LS-Sampling* with different hyper-parameter settings of λ on all benchmarks, and Table 8 shows the average execution time required by *LS-Sampling* with different hyper-parameter settings of λ on all benchmarks. Tables 7 and 8 demonstrate that, when the value of λ becomes larger, the 2-wise coverage and the 3-wise coverage achieved by *LS-Sampling* become higher, and meanwhile the execution time required by *LS-Sampling* becomes longer. The results in Tables 7 and 8 demonstrate the flexibility of *LS-Sampling*, since controlling λ can achieve a trade-off between coverage and execution time. Also, *LS-Sampling* can achieve a good balance according to our default hyper-parameter setting (*i.e.,* λ=100).

5.6 Threats to Validity

There are two potential threats to validity of our experiments:

Experimental setup for k: Recalling that k denotes the number of valid test cases, a potential threat to validity of our experiments is the value of k we set in our experiments. Following the most recent work [7], we conduct experiments with k=1,000. To reduce this threat, we also conduct experiments with k=100 as well. Our experimental results in Tables 1–3 present that *LS-Sampling* (with k=100) can still achieve higher 2-wise coverage and higher 3-wise coverage than uniform sampling (with k=1,000) and *Baital* (with k=1,000). Furthermore, we illustrate the 3-wise coverage achieved by *LS-Sampling*, uniform sampling and *Baital* with k up to 5,000 for the `financial` benchmark in Figure 1. Our experiments show that *LS-Sampling* can consistently outperforms uniform sampling and *Baital* in various scenarios under different values of k.

Generality of our experiments: In order to reduce the threat of limited generality of our experiments, we conduct our experiments to evaluate all competing approaches for solving the t-wise CovMax problem with both t=2 and t=3, while the most recent work [7] only focuses on solving the 2-wise CovMax problem. Furthermore, in our experiments we adopt a large number of benchmarks to evaluate each competing approach. Those benchmarks cover diverse numbers of variables and hard constraints, and have been extensively studied in previous work [7, 37, 45, 69, 72, 75].

6 RELATED WORK

Combinatorial interaction testing (CIT) is an important research topic in software engineering, and has been extensively studied in past decades. Comprehensive literature review can be found in the book by Kuhn *et al.* [38] and survey articles [67, 80].

In classical CIT, researchers aim to build the minimum-sized test suite with full t-wise coverage (also known as covering array) such that all possible valid t-wise option combinations in the entire configuration space are covered. Existing methods for building covering arrays include mathematical construction methods (*e.g.,* [6, 30, 86]), greedy heuristic-based methods (*e.g.,* [11–13, 16, 42–44, 81, 83, 85, 87]), and meta-heuristic search methods (*e.g.,* [11, 17–19, 22, 24, 25, 32, 46, 47, 56, 61]). In practice, achieving full t-wise coverage is difficult or even infeasible when there are large numbers of configuration options [69]. Also, it is recognized that covering arrays would possibly be large for highly configurable systems and would exceed the testing budget [7]. Hence, for testing highly configurable systems, compared to building covering arrays, an advisable solution is to solve the t-wise coverage maximum (t-wise CovMax) problem studied in this work, which aims to generate a fixed-size test suite that maximizes the t-wise coverage.

For solving the t-wise CovMax problem, previous studies present that uniform sampling, which tries to sample each valid configuration with equal probability, shows its effectiveness [7, 69]. Apart

from solving the t-wise CovMax problem, a number of research works tries to incorporate uniform sampling, as a dominant domain-agnostic method, with combinatorial solving and knowledge compilation techniques. In particular, Lopez-Herrejon *et al.* [48] evaluated the relationship between interaction coverage and the test suite size in a multi-objective way. Oh *et al.* [68] studied uniform sampling by incorporating it with binary decision diagrams. Plazar *et al.* [75] conducted extensive empirical evaluations to assess two uniform sampling tools *UniGen2* [15] and *QuickSampler* [20]. Unfortunately, uniform sampling suffers a severe issue that it cannot well handle configurable systems with complex constraints [7]. Besides uniform sampling, recently, Baranov *et al.* [7] proposed an adaptive weighted sampling approach *Baital*, which is the current state-of-the-art approach for solving the t-wise CovMax problem. The experiments in the literature [7] present that *Baital* can achieve higher 2-wise coverage than uniform sampling on most benchmarks. However, *Baital* does not optimize the objective of the t-wise CovMax problem explicitly during its sampling process, which makes it ineffective; in the meanwhile, *Baital* handles hard constraints through costly knowledge compilation techniques, which makes it inefficient.

Compared to uniform sampling and *Baital*, our *LS-Sampling* approach is based on an effective local search framework, whose search process is directly guided by the optimization objective. Also, *LS-Sampling* leverages the effectiveness of a number of core algorithmic mechanisms, including formula simplification, dynamic mechanism for updating sampling probabilities and diversity-aware heuristic search, in order to strengthen its both effectiveness and inefficiency. As reported and analyzed in Section 5, our extensive experimental results on a large number of public benchmarks collected from real-world, highly configurable systems confirm both the effectiveness and the efficiency of our *LS-Sampling* approach.

7 CONCLUSION

In this paper, we propose a novel approach dubbed *LS-Sampling* (*Local Search based Sampling*) for achieving high t-wise coverage. Extensive experiments on a large number of public benchmarks, which are collected from real-world, highly configurable software systems, demonstrate that *LS-Sampling* can achieve much higher t-wise coverage than its state-of-the-art competitors, *i.e.,* uniform sampling and *Baital*. The results indicate both the effectiveness and the efficiency of our *LS-Sampling* approach. More encouragingly, *LS-Sampling* can achieve higher 2-wise coverage and higher 3-wise coverage through generating much smaller test suits than its state-of-the-art competitors. Furthermore, *LS-Sampling* requires much less execution time to generate test suites with higher 2-wise coverage and higher 3-wise coverage compared to *Baital*.

The implementation of *LS-Sampling*, the benchmarks used in our experiments and the detailed experimental results (including the full results of the 2-wise coverage and the 3-wise coverage achieved by *LS-Sampling* and its competitors, *i.e.,* uniform sampling and *Baital*, with both k=100 and k=1,000 on all benchmarks) are available at https://github.com/chuanluocs/LS-Sampling.

REFERENCES

[1] Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. 2016. IncLing: Efficient product-line testing using incremental pairwise sampling. In *Proceedings of GPCE 2016*. 144–155.

[2] 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.

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

[4] Gilles Audemard, George Katsirelos, and Laurent Simon. 2010. A Restriction of Extended Resolution for Clause Learning SAT Solvers. In *Proceedings of AAAI 2010*. 15–20.

[5] Adrian Balint and Norbert Manthey. 2013. Boosting the Performance of SLS and CDCL Solvers by Preprocessor Tuning. In *In Proceedings of POS 2013*. 1–14.

[6] 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.

[7] 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.

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

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

[10] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. 2021. Preprocessing in SAT Solving. In *Handbook of Satisfiability*. 391–435.

[11] 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.

[12] 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.

[13] 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.

[14] Sooyoung Cha and Hakjoo Oh. 2019. Concolic testing with adaptively changing search heuristics. In *Proceedings of ESEC/FSE 2019*. 235–245.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient sampling of SAT solutions for testing. In *Proceedings of ICSE 2018*. 549–559.

[21] Niklas Eén and Armin Biere. 2005. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of SAT 2005*. 61–75.

[22] 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.

[23] 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.

[24] Syed A. Ghazi and Moataz A. Ahmed. 2003. Pair-wise test coverage using genetic algorithms. In *Proceedings of CEC 2003*. 1420–1424.

[25] 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.

[26] 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.

[27] Mark Harman and Phil McMinn. 2010. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247.

[28] Ruben Heradio, David Fernández-Amorós, José A. Galindo, and David Benavides. 2020. Uniform and scalable SAT-sampling for configurable systems. In *Proceedings of SPLC 2020*. 17:1–17:11.

[29] Marijn Heule, Matti Järvisalo, and Armin Biere. 2010. Clause Elimination Procedures for CNF Formulas. In *Proceedings of LPAR 2010*. 357–371.

[30] 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.

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

[32] 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.

[33] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. 2012. Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines. In *Proceedings of MODELS 2012*. 269–284.

[34] 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.

[35] Henry Kautz, Ashish Sabharwal, and Bart Selman. 2021. Incomplete Algorithms. In *Handbook of Satisfiability*. 185–203.

[36] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2016. SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence* 232 (2016), 20–42.

[37] 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.

[38] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2013. *Introduction to combinatorial testing*. CRC press.

[39] 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.

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

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

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[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] Inês Lynce and João P. Marques Silva. 2003. Probing-Based Preprocessing Techniques for Propositional Satisfiability. In *Proceedings of ICTAI 2003*. 105–110.

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

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

[62] 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.

[63] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Proceedings of SPLC 2009*. 231–240.

[64] 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.

[65] 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.

[66] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2012. Preprocessing in Incremental SAT. In *Proceedings of SAT 2012*. 256–269.

[67] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *Comput. Surveys* 43, 2 (2011), 11:1–11:29.

[68] 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.

[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] Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham (Eds.). 2013. *Handbook of Combinatorial Optimization*. Springer.

[71] 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.

[72] 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.

[73] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. 2008. Vivifying Propositional Clausal Formulae. In *Proceedings of ECAI 2008*. 525–529.

[74] Florian Pigorsch and Christoph Scholl. 2010. An AIG-Based QBF-solver using SAT for preprocessing. In *Proceedings of DAC 2010*. 170–175.

[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] 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.

[77] Shubham Sharma, Rahul Gupta, Subhajit Roy, and Kuldeep S. Meel. 2018. Knowledge Compilation meets Uniform Sampling. In *Proceedings of LPAR 2018*. 620–636.

[78] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. 2004. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances. In *Proceedings of SAT 2004*. 276–291.

[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. *Comput. 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] 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.

[83] Ziyuan Wang, Changhai Nie, and Baowen Xu. 2007. Generating combinatorial test suite for interaction relationship. In *Proceedings of SOQUA 2007*. 55–61.

[84] Thomas Weise, Zijun Wu, and Markus Wagner. 2019. An Improved Generic Bet-and-Run Strategy with Performance Prediction for Stochastic Local Search. In *Proceedings of AAAI 2019*. 2395–2402.

[85] 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.

[86] 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.

[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.