Generating Pairwise Covering Arrays for Highly Configurable Software Systems

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

Jianping Song
School of Software,
Beihang University
Beijing, China
19231112@buaa.edu.cn

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

Yibei Li
Department of Physics,
Tsinghua University
Beijing, China
yibeili@tsinghua.edu.cn

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

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

ABSTRACT

Highly configurable software systems play crucial roles in real-world applications, which urgently calls for useful testing methods. Combinatorial interaction testing (CIT) is an effective methodology for detecting those faults that are triggered by the interaction of any t options, where t is the testing strength. Pairwise testing, *i.e.,* CIT with $t = 2$, is known to be the most practical and popular CIT technique, and the pairwise covering array generation (PCAG) problem is the most critical problem in pairwise testing. Due to the practical importance of PCAG, many PCAG algorithms have been proposed. Unfortunately, existing PCAG algorithms suffer from the severe scalability problem. To this end, the SPLC Scalability Challenge (*i.e.,* Product Sampling for Product Lines: The Scalability Challenge) has been proposed since 2019, in order to motivate researchers to develop practical PCAG algorithms for overcoming this scalability problem. In this work, we present a practical PCAG algorithm dubbed *SamplingCA-ASF*. To the best of our knowledge, our experiments show that *SamplingCA-ASF* is the first algorithm that can generate PCAs for `Automotive02` and `Linux`, the two hardest and largest-scale instances in the SPLC Scalability Challenge, within reasonable time. Our experimental results indicate that *SamplingCA-ASF* can effectively alleviate the scalability problem in pairwise testing.

CCS CONCEPTS

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

KEYWORDS

pairwise testing, scalability problem, covering array generation

ACM Reference Format:
Chuan Luo, Jianping Song, Qiyuan Zhao, Yibei Li, Shaowei Cai, and Chunming Hu. 2023. Generating Pairwise Covering Arrays for Highly Configurable Software Systems. In *27th ACM International Systems and Software Product Line Conference - Volume A (SPLC '23), August 28-September 1, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3579027.3608998

1 INTRODUCTION

To satisfy the increasing demands for software customization, the development of highly configurable software systems is essential in real-world applications and hence draws much attention from academia and industry [5, 19, 28–33, 42]. Highly configurable software systems provide practitioners with many configuration options, so practitioners can configure software systems to meet their practical requirements [1, 36]. However, it is recognized that effectively testing such highly configurable software systems is difficult, since the number of possible configurations grows exponentially with the increment in the number of option [30–32]. For example, given a configurable software system with 100 options, where each option has 2 possible values, there can be 2^{100} possible configurations to test. As a result, testing all possible configurations is impractical in real-world applications.

Combinatorial interaction testing (CIT) is a practical and effective methodology for disclosing the option-interaction related faults caused by combinations of any t options, where t is the testing strength [30, 32, 39, 52]. Pairwise testing, *i.e.,* CIT with $t = 2$, is the most popular CIT approach in practice [22, 23, 32]. Compared to CIT with large values of t, pairwise testing can construct a test suite of much small size while keeping high capability of fault identification. Also, a number of empirical studies [22, 23] on highly configurable software systems demonstrate that pairwise testing is able to detect the majority of faults, which confirms the effectiveness of pairwise testing in practical scenarios.

For practical configurable software systems, there usually exist hard constraints on options, such as functional dependencies and exclusiveness. Each adopted test case needs to satisfy all constraints, since using a test case that violates any constraint would incur inaccurate testing outcome [43]. A test case is valid if it satisfies all constraints. A pairwise tuple, also known as a 2-wise tuple, is a combination of values of exactly two options, and a pairwise tuple is valid if it is covered by at least one valid test case. The task of pairwise testing is to generate a pairwise covering array (PCA), which is a set of valid test cases covering all valid pairwise tuples. Since the size of PCA (*i.e.,* the number of valid test cases in PCA) directly affects the testing budget, the most essential problem in pairwise testing is the problem of pairwise covering array generation (PCAG), which aims to build a PCA of minimum size.

Because of the importance of PCAG, many PCAG algorithms have been developed (*e.g.,* [2, 4, 7, 7–18, 21, 24–26, 28–30, 32, 35, 47, 48, 50, 51, 53–55]). However, existing PCAG algorithms suffer from the serious scalability problem [44, 49]. That is, when dealing with highly configurable software systems that expose many options, existing PCAG algorithms usually produce test suites of unacceptably large sizes, do not terminate, or even fail to generate test suites in reasonable time [44, 49]. Since the scalability problem is of high severity, the SPLC Scalability Challenge (*i.e.,* Product Sampling for Product Lines: The Scalability Challenge) [44] has been proposed since 2019. This SPLC Scalability Challenge presents two challenging PCAG instances, both of which are collected from highly configurable software systems with tens of thousands of configuration options, *i.e.,* `Automotive02` and `Linux`. These two highly configurable software systems are known to play crucial roles in real-world applications, which urgently calls for practical solutions. Unfortunately, to the best of our knowledge, in the literature there is no existing PCAG algorithm that could generate PCAs within reasonable time (*e.g.,* 1 day) for `Automotive02` and `Linux`.

In this work, we present a practical sampling-based PCAG algorithm dubbed *SamplingCA-ASF*. In fact, *SamplingCA-ASF* is a variant of our recently-proposed algorithm named *SamplingCA* [32] with several modifications. To demonstrate its effectiveness, we perform experiments to evaluate *SamplingCA-ASF* on `Automotive02` and `Linux`. Our experiments present that *SamplingCA-ASF* generates a PCA of 58058 test cases for `Automotive02` with the running time of 42842.6 seconds, and *SamplingCA-ASF* builds a PCA of 2561 test cases for `Linux` with the running time of 65335.6 seconds.

Our main contributions of this work are summarized as follows.

- We present a practical PCAG algorithm dubbed *SamplingCA-ASF*, which can effectively mitigate the scalability problem.
- Our experiments demonstrate that *SamplingCA-ASF* can successfully build PCAs of acceptable sizes for two challenging instances `Automotive02` and `Linux` within reasonable running time. To the best of our knowledge, *SamplingCA-ASF* is the first PCAG algorithm that can generate PCAs for `Automotive02` and `Linux` within reasonable running time.

2 PRELIMINARIES

In this section, we provide necessary preliminaries about this work.

2.1 Pairwise Covering Array Generation

We introduce necessary definitions about PCAG as follows.

System Under Test: A system under test S, also known as a configurable software system and an instance in this work, is able to be expressed as a pair $S = (O, H)$, where notations O and H stand for a set of options and a collection of hard constraints, respectively.

Without loss of generality, following recent studies that concentrate on testing highly configurable software systems [5, 31, 32, 41, 44], in this work we focus on the binary scenario where each option has two possible values. For testing highly configurable software systems, it has been widely accepted that the general scenario, where each option has multiple possible values, can be transformed into the binary scenario (*i.e.,* the one analyzed in this work) in an effective way [5, 31, 32, 41, 44]. Moreover, the two challenging PCAG instances (*i.e.,* `Automotive02` and `Linux`) evaluated in this work are of binary scenario, and both of them are encoded from the general scenario and are collected from real-world, highly configurable software systems. Therefore, the analysis of the binary scenario is of significant importance in practice [32].

Test Case: Given an SUT $S = (O, H)$, a test case tc, also known as a configuration, is a collection of $|O|$ pairs, *i.e.,* $tc = \{(o_i, r_i) \mid i \text{ is an integer, and } 1 \leq i \leq |O|\}$, implying that each option o_i is assigned value r_i. Due to the existence of hard constraints, a test case tc is valid if tc satisfies all hard constraints in H; otherwise, tc is invalid. A test suite is a set of test cases.

Pairwise Tuple: A pairwise tuple is a combination of the values of two options. For example, the pairwise tuple $\tau = \{(o_i, r_i), (o_j, r_j)\}$ indicates that options o_i and o_j are assigned values r_i and r_j, respectively. Given a pairwise tuple τ a test case tc, pairwise tuple τ is covered by test case tc if $\tau \subseteq tc$. Also, given a pairwise tuple τ and a test suite A, pairwise tuple τ is covered by A if τ is covered by at least one test case in A. Furthermore, a pairwise tuple τ is valid if τ is covered by at least one valid test case; otherwise, τ is invalid.

Pairwise Covering Array: A pairwise covering array (PCA) T is a test suite, which is comprised of valid test cases and covers all valid pairwise tuples. In another word, a PCA T is a set of valid test cases such that all valid pairwise tuples are covered by T.

Pairwise Covering Array Generation: Given an SUT, the problem of PCA generation (PCAG) is to construct a PCA as small as possible. The PCAG problem is the most essential problem in pairwise testing, and it is a challenging combinatorial optimization problem [26, 40]. Hence, developing effective PCAG algorithms is important.

2.2 Boolean Formulae

It is acknowledged that an SUT (*i.e.,* a configurable software system) is related to a Boolean formula; particularly, an SUT can be effectively modeled as a Boolean formula [3, 5, 6, 31, 32, 36, 41, 44, 45]. Actually, we can deal with a highly configurable system ineffectively via analyzing the modeled Boolean formula [27, 38]. Hence, this subsection describes key concepts about Boolean formula and discusses the connection between SUT and Boolean formula.

Given a Boolean variable x, the value domain of x is $\{0, 1\}$. Actually, Boolean variables are fundamental elements of Boolean formulae. A Boolean variable x_i and its negative form $\neg x_i$ are literals, and a disjunction of literals can construct a clause c_j, *i.e.,* $c_j = x_{j,1} \vee x_{j,2} \vee \cdots \vee x_{j,k_j}$, where k_j is the length of c_j. Given a collection of n Boolean variables, a Boolean formula F in conjunctive normal form (CNF) is a conjunction of m clauses, *i.e.,* $F = c_1 \wedge c_2 \cdots c_m$. Given a Boolean formula F in CNF, notation $V(F)$ represents the set of n Boolean variables (*i.e.,* $|V(F)| = n$), and notation $C(F)$ stands for the set of m clauses (*i.e.,* $|C(F)| = m$).

Algorithm 1: Skeleton of *SamplingCA* Algorithm ([32])

```
Input: $F$: Boolean formula in CNF;
Output: $T$: pairwise covering array (PCA) of $F$;
1  $\alpha \leftarrow$ generate the first, valid test case;
2  $T \leftarrow \{\alpha\}$;
3  while True do
4      $C \leftarrow$ construct a candidate collection of valid test cases;
5      $\beta^* \leftarrow$ select the test case with the largest gain from $C$;
6      if gain of $\beta^*$ is not greater than 0 then break;
7      $T \leftarrow T \cup \{\beta^*\}$;
8  foreach possible pairwise tuple $\tau$ of $F$ do
9      if $\tau$ is not covered by any test case in $T$ then
10         Calling SAT solver to justify the validity status of $\tau$;
11         if $\tau$ is a valid pairwise tuple then
12             $tc \leftarrow$ a test case covering $\tau$ by calling SAT solver;
13             $T \leftarrow T \cup \{tc\}$;
14 return $T$;
```

For a Boolean Formula F in CNF, a mapping $\alpha : V(F) \longrightarrow \{0, 1\}$ is an assignment of F. Given an assignment α, α is a complete assignment if α maps all variables in $V(F)$. Given a clause c_j and a complete assignment α, clause c_j is satisfied if at least one literal in c_j evaluates to 1 under α; otherwise, c_j is unsatisfied. Given a formula F and a complete assignment α, if α satisfies all clauses in $C(F)$, α is a satisfying assignment (also known as a solution) of F.

Given an SUT $S = (O, H)$ and the Boolean Formula F that is modeled from S, it is straightforward that the option set O and the constraint set H in SUT S correspond to the variable set $V(F)$ and the clause set $C(F)$ of formula F, respectively. A satisfying assignment of F is a valid test case of S. Also, a pairwise tuple of S is related to a combination of F's two literals; for example, the pairwise tuple $\{(o_i, 1), (o_j, 0)\}$ of S is related to a combination of F's two literals $\{x_i, \neg x_j\}$. Actually, given an SUT S and the Boolean formula F that is modeled from S, the PCAG problem can be understood as finding a collection of F's satisfying assignments, such that all valid pairwise tuples of S are covered.

In theory, given a Boolean formula F, the problem of finding one satisfying assignment of F is indeed the well-known, Boolean satisfiability (SAT) problem, which is an NP-complete problem. Hence, a SAT solver (*i.e.,* a practical algorithm for solving the SAT problem) is required in PCAG algorithms. In fact, our *SamplingCA-ASF* algorithm proposed in this work also invokes an effective SAT solver called *ContextSAT* [32]. For technical details of *ContextSAT*, readers can refer to the literature [32].

3 OUR *SAMPLINGCA-ASF* ALGORITHM

This section presents the *SamplingCA-ASF* algorithm, which can effectively generate PCAs for highly configurable software systems.

3.1 Review of Existing *SamplingCA* Algorithm

As described in Section 1, our *SamplingCA-ASF* algorithm proposed in this work is actually a variant of our previous PCAG algorithm called *SamplingCA* [32], which exhibit state-of-the-art performance in PCAG solving. Since *SamplingCA* serves as a basis of *SamplingCA-ASF*, in this subsection we briefly review the *SamplingCA* algorithm. For more algorithmic details and technical discussions about the *SamplingCA* algorithm, readers can refer to the literature [32].

The skeleton of our previous *SamplingCA* algorithm [32] is summarized in Algorithm 1, while the whole pseudo code of *SamplingCA* is outlined in the literature [32]. The input of *SamplingCA* is a Boolean formula in CNF, denoted by F, and the output of *SamplingCA* is a PCA of F, denoted by T. There are three phases in *SamplingCA*, *i.e.,* initialization phase (Lines 1 and 2 in Algorithm 1), sampling phase (Lines 3–7 in Algorithm 1), and full covering phase (Lines 8–13 in Algorithm 1). All these three phases are briefly described in the following paragraphs.

3.1.1 *The Initialization Phase of* SamplingCA.

In the initialization phase, *SamplingCA* aims to generate the first valid test case α and initializes the test suite T as the collection including α. We would like to note that in the following phases (*i.e.,* the sampling phase and the full covering phase), test suite T would be repeatedly expanded through adding newly-generated, valid test cases until T covers all valid pairwise tuples (*i.e.,* T becomes a PCA). After all phases are performed, T is the final PCA output by *SamplingCA*.

3.1.2 *The Sampling Phase of* SamplingCA.

In the sampling phase, *SamplingCA* works in an iterative manner, where in each iteration a valid test case is constructed and added into test suite T. In each iteration, *SamplingCA* works as follows. First, *SamplingCA* constructs a candidate collection C consisting of valid test cases. For each test case β in C, β is sampled from the entire space of test cases; after sampling, if β is invalid, β is altered to be a valid test through invoking a SAT solver. Then, *SamplingCA* tries to select a valid test case from candidate collection C and adds the chosen, valid test case into test suite T. To accomplish the selection process, *SamplingCA* defines an important evaluation metric called *gain* [32], so as to quantify the benefit of a test case if it is added into T. According to the literature [32], given a test suite T and a valid test case $\beta \notin T$, the *gain* of β with regard to T is the increment in the number of covered pairwise tuples if β is added into T. After candidate collection C is constructed, *SamplingCA* selects the test case β^* with the largest value of *gain* from C. Through this way, the addition of β^* into T would increase the number of covered pairwise tuples as much as possible. Once the *gain* of the selected test case β^* is not greater than 0, the sampling phase terminates.

3.1.3 *The Full Covering Phase of* SamplingCA.

Since the test suite returned by the sampling phase does not guarantee to cover all valid pairwise tuples, in the full covering phase, *SamplingCA* focuses on adding a number of valid test cases into T in order to make T become a PCA. Specifically, in the full covering phase, *SamplingCA* enumerates all possible pairwise tuples and checks the validity status of each possible pairwise tuple τ: if τ is a valid pairwise tuple and is not covered by T, then *SamplingCA* would call a SAT solver to generate a valid test case covering τ and adds the generated, valid test case into T. In this manner, the test suite returned by the full covering phase, which is also the final test suite output by the entire *SamplingCA* algorithm, is ensured to be a PCA.

3.1.4 *Summary of* SamplingCA.

As described above, it is apparent that the sampling phase and the full covering play crucial roles in the process of PCA generation, and their effectiveness directly impacts the practical performance of the *SamplingCA* algorithm.

3.2 Modifications to *SamplingCA* Algorithm

The empirical results reported in the literature [32] present that *SamplingCA* can generate PCAs for configurable software systems with around 1000 options. However, in our preliminary experiments, we try to use *SamplingCA* to generate PCAs for `Automotive02` and `Linux`, the two hardest and largest-scale instances presented by the SPLC Scalability Challenge [44], and our preliminary results show that *SamplingCA* cannot generate PCAs for these two instances within reasonable time (*e.g.,* 1 day). This is not surprising, since `Automotive02` and `Linux` provide more than 18000 and 76000 options, respectively, and their problem scales are possibly beyond the problem scale that *SamplingCA* can handle.

In this work, we make modifications to *SamplingCA*, resulting in a variant of *SamplingCA* dubbed *SamplingCA-ASF*, so as to generate PCAs for those two challenging instances (*i.e.,* `Automotive02` and `Linux`). In particular, we modify the sampling phase and the full covering phase of the original *SamplingCA* algorithm, and the main differences between *SamplingCA* and *SamplingCA-ASF* are highlighted in blue color in Algorithm 1.

3.2.1 *Modification to the Sampling Phase of* SamplingCA.

As discussed in Section 3.1.2, the evaluation metric of *gain* plays a critical role in the sampling phase. Particularly, the time complexity of calculating the *gain* of a given test case greatly affects the efficiency of the sampling phase. According to the implementation of *SamplingCA*,[1] the *gain* value of β equals to the number of valid pairwise tuples that are covered by β and are meanwhile not covered by T. In this way, calculating β's *gain* value needs to traverse all the valid pairwise tuples that covered by β, and hence the time complexity of computing *gain* for a given test case is $O(n^2)$, where n denotes the number of options. As aforementioned, `Automotive02` and `Linux` have more than 18000 and 76000 options, respectively, the operation of calculating *gain* needs a certain amount of running time due to the time complexity of $O(n^2)$. Moreover, *SamplingCA*'s process of PCA generation requires to invoke the operation of calculating *gain* many times [32], which possibly explains the reason why *SamplingCA* cannot generate PCAs for `Automotive02` and `Linux` within reasonable time.

In order to mitigate this issue, we replace the above method of *gain* calculation with the approximated scoring function (ASF) proposed in our recent study [33]. According to the discussion in our recent work [33], different from the *gain* calculation method adopted by *SamplingCA*, the ASF method does not calculate the exact *gain* value, and it aims to computes an estimated value of *gain* in a much efficient way. According to our recent work [33], the ASF method works as follows. Before entering the sampling phase, a measuring set M consisting of pairwise tuples, which are randomly sampled from the entire space of possible test cases, is constructed. When quantifying the benefit of a given test case β, the ASF method estimates β's *gain* value as the increment in the number of covered pairwise tuples that belong to measuring set M. As discussed and evaluated in the literature [33], the ASF method not only approximates the exact *gain* value effectively, but also runs much more efficient that the method of calculating the exact *gain* value. Hence, *SamplingCA-ASF* is highly efficient through incorporating the ASF method.

3.2.2 *Modification to the Full Covering Phase of* SamplingCA.

The test suite T returned by the sampling phase is not ensured to cover all valid pairwise tuples, so there usually exist a number of remaining valid pairwise tuples that are not covered by T. As introduced in Section 3.1.3 and presented in Lines 8–13 in Algorithm 1, in the full covering phase *SamplingCA* traverses each possible pairwise tuple τ to build the remaining set of all valid pairwise tuples that are not covered by T. For each uncovered, valid pairwise tuple τ, *SamplingCA* calls a SAT solver to generate a valid test case tc that covers τ and then adds tc into T. This method of generating one test case that only targets to cover a single, valid pairwise tuple would possibly result in a PCA of large size.

It is known that adopting a large-sized PCA to test highly configurable software systems would cost a large amount of testing budget in practice [31–33] and the ultimate objective of the PCAG problem is to find a PCA of minimum size, so it is crucial to reduce the size of generated PCA. Hence, in order to mitigate this severe issue, we enhance the full covering phase as follows. Compared to *SamplingCA*'s original full covering phase that targets to generate a valid test case for each uncovered, valid pairwise tuple, the modified full covering phase underlying *SamplingCA-ASF* performs a repetitive process. In each step of the repetitive process, *SamplingCA-ASF* aims to construct a valid test case to cover multiple uncovered, valid pairwise tuples, rather than just one single uncovered, valid pairwise tuple. It is intuitive that through this method *SamplingCA-ASF* is able to reduce the number of valid test cases added in the full covering phase. As a result, compared to the original *SamplingCA* algorithm, *SamplingCA-ASF* is able to generate smaller-sized PCAs.

4 EXPERIMENTS

In this section, we conduct experiments to evaluate the practical performance of *SamplingCA-ASF* to generate PCAs for two highly configurable software systems, *i.e.,* `Automotive02` and `Linux`. Particularly, we first describe two challenging PCAG instances adopted in our experiments, and introduce the state-of-the-art competitor. Then, we present the research question and the experimental setup. Finally, we report and analyze the experimental results.

4.1 Challenging PCAG Instances

As aforementioned, there are two challenging PCAG instances that are adopted in our experiments, *i.e.,* `Automotive02` and `Linux`, both of which are taken from the SPLC Scalability Challenge and are the hardest and largest-sized instances in the SPLC Scalability Challenge [44].

The `Automotive02` instance is publicly available online,[2] and we use its latest version (*i.e.,* Version: V4) in our experiments. Since our *SamplingCA-ASF* algorithm is designed to process Boolean formula, in our experiments the `Automotive02` instance is encoded as a Boolean formula through the *FeatureIDE* platform[3] [37, 46]. After

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

[2]https://github.com/PettTo/SPLC2019_The-Scalability-Challenge_Product-Lines/blob/master/Automotive02

[3]https://featureide.github.io/

transforming into a Boolean formula, the `Automotive02` instance has 18616 options and 350119 constraints.

The `Linux` instance is also publicly available online,[4] and we adopt its newest version (*i.e.,* Version: 2018-01-14T09_51_25-08_00) in our experiments. The `Linux` instance has been encoded as a Boolean formula[5] by the organizers of the SPLC Scalability Challenge, so we directly utilize this Boolean formula. The `Linux` instance in our experiments has 76815 options and 774148 constraints.

For the above procedures for encoding and obtaining the Boolean formulae of the `Automotive02` and `Linux` instances, we have consulted the organizers of the SPLC Scalability Challenge [44], and the organizers have kindly confirmed the correctness of our procedures. We note that both Boolean formulae of the `Automotive02` and `Linux` instances are publicly available at our public repository.[6]

4.2 State-of-the-art PCAG Competitor

In order to demonstrate the effectiveness of our *SamplingCA-ASF* algorithm proposed in this work, we adopt the original *SamplingCA* algorithm [32] as the state-of-the-art competitor of *SamplingCA-ASF* in our experiments. *SamplingCA* [32] has bee briefly reviewed in Section 3.1, and it is a recently-proposed, effective algorithm and exhibits state-of-the-art performance in PCAG solving. As reported in the literature [32], the performance of *SamplingCA* is much better than that of other PCAG algorithms when generating PCAs for highly configurable software systems. Besides the high performance, *SamplingCA* serves as the algorithmic basis for *SamplingCA-ASF*, so it is instructive to conduct the comparison between *SamplingCA* and *SamplingCA-ASF*. The source code of *SamplingCA* is publicly available online.[1]

4.3 Research Question

To the best of our knowledge, there is no existing PCAG algorithm that can generate PCAs within reasonable time for those two hardest and largest-sized instances in the SPLC Scalability Challenge [44], *i.e.,* `Automotive02` and `Linux`. Hence, in order to examine whether our *SamplingCA-ASF* algorithm alleviates the severe scalability problem, our evaluation of *SamplingCA-ASF* concentrates on answering the following research question (RQ).

RQ: Can *SamplingCA-ASF* generate PCA for these two challenging instances, *i.e.,* `Automotive02` and `Linux`, within reasonable running time?

In this RQ, we perform experiments to analyze whether our *SamplingCA-ASF* algorithm can generate PCAs for `Automotive02` and `Linux` within acceptable running time.

4.4 Experimental Setup

All experiments in this work are conducted on a computing machine equipped with AMD EPYC 7763 CPU AND 1TB memory, and the operating system installed on that computing machine is Ubuntu 20.04.4 LTS.

As described before, *SamplingCA-ASF* is a variant of *SamplingCA*, and we implement *SamplingCA-ASF* based on the source code of *SamplingCA* [32]. Because the source code of *SamplingCA* adopts a Boolean formula preprocessing tool called *Coprocessor* [34] to equivalently simplify the Boolean formulae of both `Automotive02` and `Linux` instances, our *SamplingCA-ASF* algorithm also activates the same preprocessing tool to simplify these two instances. We would like to note that the implementation of *SamplingCA-ASF* is publicly available at our public repository.[6] For both competing algorithms (*i.e., SamplingCA-ASF* and *SamplingCA*), each competing algorithm is performed one run per PCAG instance, with a cutoff time of 86400 seconds (*i.e.,* 1 day). In order to make our comparison fair, we use the same hyper-parameter settings for both *SamplingCA-ASF* and *SamplingCA*, recommended by the literature [32]. For each competing algorithm on each PCAG instance, we report the size of the generated PCA, denoted by 'size', and the running time, denoted by 'time'. In our experiments, the running time is measured in second. Furthermore, if a competing algorithm fails to generate a PCA for a PCAG instance within the cutoff time, the corresponding results of both 'size' and 'time' are marked as '–' in our experiments.

Table 1: Results of *SamplingCA-ASF* and *SamplingCA* on the challenging `Automotive02` and `Linux` instances.

Instance	*SamplingCA-ASF*		*SamplingCA*	
	size	time (sec)	size	time (sec)
`Automotive02`	**58058**	**42842.6**	–	–
`Linux`	**2561**	**65335.6**	–	–

4.5 Experimental Results

Table 1 presents the comparative results of *SamplingCA-ASF* and *SamplingCA* on those two challenging `Automotive02` and `Linux` instances. According to the experimental results reported in Table 1, it is clear that *SamplingCA-ASF* is able to generate PCAs within the cutoff time for both challenging `Automotive02` and `Linux` instances, while *SamplingCA* fails to construct PCAs within the cutoff time for both of them. As discussed in Section 3.2, it is not surprising that *SamplingCA* fails to produce PCAs for both challenging instances, because both `Automotive02` and `Linux` instances are of large scale (*i.e.,* the `Automotive02` instance has 18616 options, and the `Linux` instance has 76815 options), and their scales possibly exceed the problem scale that the original *SamplingCA* algorithm can process.

According to Table 1, *SamplingCA-ASF* takes 42842.6 seconds to construct a PCA consisting of 58058 test cases for the `Automotive02` instance, and it costs 65335.6 seconds to generate a PCA containing 2561 test cases for the `Linux` instance. Our experiments clearly present the superiority of *SamplingCA-ASF* over *SamplingCA* on solving these two challenging instances. More encouragingly, to the best of our knowledge, currently there is no PCAG algorithm that can generate PCAs for `Automotive02` and `Linux` within reasonable time, indicating that *SamplingCA-ASF* is able to effectively alleviate the scalability problem in pairwise testing.

Also, we further analyze the quality of the test case generated by *SamplingCA-ASF*. Before discussing the results, we first introduce a useful evaluation metric called pairwise coverage [5, 31]. In the

[4]https://github.com/PettTo/Feature-Model-History-of-Linux
[5]https://github.com/PettTo/Feature-Model-History-of-Linux/blob/master/2018/2018-01-14T09_51_25-08_00/out.dimacs.zip
[6]https://github.com/chuanluocs/SamplingCA-ASF

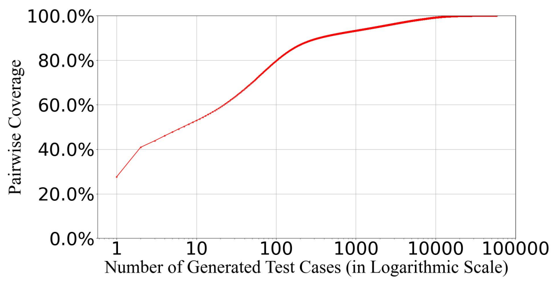

Figure 1: Pairwise coverage achieved by *SamplingCA-ASF* with different numbers of test cases on `Automotive02`.

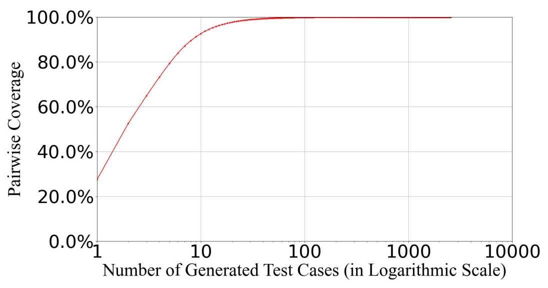

Figure 2: Pairwise coverage achieved by *SamplingCA-ASF* with different numbers of test cases on `Linux`.

context of testing highly configurable software systems, pairwise coverage is broadly recognized to be a powerful evaluation metric to assess the quality of a test suite, and it plays a crucial role in this empirical analysis [5, 31]. Given an SUT S and a test suite T, the pairwise coverage of T is computed as the ratio between the number of valid pairwise tuples covered by T and the total number of all possible, valid pairwise tuples for the given SUT S. Specifically, if a test suite T is a PCA, then it achieves the full pairwise coverage (*i.e.,* the pairwise coverage of 100%). It is intuitive that a test suite with higher pairwise coverage indicates stronger capability of fault detection. We illustrate the pairwise coverage achieved by *SamplingCA-ASF* through generating different numbers of test cases for the `Automotive02` and `Linux` instances in Figures 1 and 2, respectively. For `Automotive02`, compared to generating 58058 test cases for achieving full pairwise coverage, *SamplingCA-ASF* only needs to construct 345, 1901 and 9055 test cases to obtain the pairwise coverage of 90%, 95% and 99%, respectively. Similarly, for `Linux`, compared to constructing 2561 test cases for achieving full pairwise coverage, *SamplingCA-ASF* only needs to build 9, 13 and 34 test cases to obtain the pairwise coverage of 90%, 95% and 99%, respectively. Our results clearly demonstrate that our *SamplingCA-ASF* algorithm is practical, since it can build a test suite of small size while preserving high quality.

5 RELATED WORK

Combinatorial interaction testing (CIT) is an important research direction in software testing, and has been widely studied in past decades. Pairwise testing, *i.e.,* CIT with $t = 2$, is recognized to be a popular and powerful CIT technique for testing highly configurable software systems [22, 23, 32]. The PCAG problem is the most essential problem in pairwise testing, and a large amount of research effort has been made to develop effective PCAG algorithms [2, 4, 7, 7–18, 21, 24–26, 28–30, 32, 35, 47, 48, 50, 51, 53–55]). However, existing PCAG algorithms suffer from the scalability problem and cannot process large-sized PCAG instances in an effective way.

Due to the high severity of the scalability problem, the SPLC Scalability Challenge [44] has been organized since 2019. The SPLC Scalability Challenge presents two difficult instances, which are collected from two important, highly configurable software systems, *i.e.,* `Automotive02` and `Linux`. To the best of our knowledge, prior to this work, there are two solutions [20, 41] submitted to the SPLC Scalability Challenge. For these two submitted solutions [20, 41], their authors evaluate the performance of a uniform sampling algorithm called *Smarch* [41] and a t-wise sampling algorithm named *YASA* [21] to generate PCAs for large-scale PCAG instances. However, *Smarch* [41] does not guarantee its output test suite to be a PCA, and the experiments in the solution paper [41] report that the test suite generated by *Smarch* obtains low pairwise coverage for highly configurable software systems. On the other hand, the work describing *YASA* [21] indicates that *YASA* fails to generate PCAs for the challenging `Automotive02` and `Linux` instances with the cutoff time of 5 days.

Compared to existing PCAG algorithms and previous solutions, to the best of our knowledge, the *SamplingCA-ASF* algorithm proposed in this work is the first algorithm that can generate PCAs for `Automotive02` and `Linux` within reasonable time, indicating that *SamplingCA-ASF* can effectively alleviate the scalability problem.

6 CONCLUSION

This work presents a practical PCAG algorithm called *SamplingCA-ASF*, a variant of our recently-proposed, state-of-the-art PCAG algorithm named *SamplingCA* with several modifications. We perform experiments to evaluate *SamplingCA-ASF* on two challenging PCAG instances, *i.e.,* `Automotive02` and `linux`. To the best of our knowledge, there is no existing PCAG algorithm that can effectively generate PCAs for `Automotive02` and `linux` within reasonable time. Encouragingly, our experiments demonstrate that *SamplingCA-ASF* can generate PCAs for `Automotive02` and `linux` within reasonable time, indicating that *SamplingCA-ASF* is able to effectively alleviate the well-known scalability problem.

DATA AVAILABILITY STATEMENT

The implementation of *SamplingCA-ASF*, both Boolean formulae of the two challenging `Automotive02` and `Linux` instances, and *SamplingCA-ASF*'s both generated PCAs for the `Automotive02` and `Linux` instances are publicly available at our public repository: **https://github.com/chuanluocs/SamplingCA-ASF**.

ACKNOWLEDGMENTS

This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4502003, in part by the National Natural Science Foundation of China under Grant 62202025 and Grant 62122078, and in part by CCF-Huawei Populus Grove Fund.

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

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

[4] 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.

[5] 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.

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

[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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Sebastian Krieter. 2020. Large-scale T-wise interaction sampling using YASA. In *Proceedings of SPLC 2020.* 29:1–29:4.

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

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] Chuan Luo, Qiyuan Zhao, Shaowei Cai, Hongyu Zhang, and Chunming Hu. 2022. SamplingCA: Effective and Efficient Sampling-Based Pairwise Testing for Highly Configurable Software Systems. In *Proceedings of ESEC/FSE 2022.* 1185–1197.

[33] Chuan Luo, Qiyuan Zhao, Jianping Song, Binqi Sun, Junjie Chen, Hongyu Zhang, and Jinkun Lin. 2023. Improving t-wise Coverage via Effective and Efficient Local Search-based Sampling. (2023).

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

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

[36] 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.

[37] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE.* Springer.

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

[39] 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.

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

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

[42] 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.

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

[44] 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.

[45] 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.

[46] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming* 79 (2014), 70–85.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] Qiyuan Zhao, Chuan Luo, Shaowei Cai, Wei Wu, Jinkun Lin, Hongyu Zhang, and Chunming Hu. 2023. CAmpactor: A Novel and Effective Local Search Algorithm for Optimizing Pairwise Covering Arrays. In *Proceedings of ESEC/FSE 2023.*