AutoCCAG: An Automated Approach to Constrained Covering Array Generation

Chuan Luo*, Jinkun Lin†, Shaowei Cai†, Xin Chen*‡, Bing He*‡, Bo Qiao*, Pu Zhao*,
Qingwei Lin*, Hongyu Zhang§, Wei Wu¶, Saravanakumar Rajmohan†, Dongmei Zhang*
*Microsoft Research, China
†State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
‡Microsoft 365, United States
§The University of Newcastle, Australia
¶L3S Research Center, Leibniz University Hannover, Germany
{chuan.luo, v-xich15, v-hebi, boqiao, puzhao, qlin, saravar, dongmeiz}@microsoft.com,
{jkunlin, william.third.wu}@gmail.com, caisw@ios.ac.cn, hongyu.zhang@newcastle.edu.au

***Abstract*—Combinatorial interaction testing (CIT) is an important technique for testing highly configurable software systems with demonstrated effectiveness in practice. The goal of CIT is to generate test cases covering the interactions of configuration options, under certain hard constraints. In this context, constrained covering arrays (CCAs) are frequently used as test cases in CIT. Constrained Covering Array Generation (CCAG) is an NP-hard combinatorial optimization problem, solving which requires an effective method for generating small CCAs. In particular, effectively solving t-way CCAG with $t \geqslant 4$ is even more challenging. Inspired by the success of automated algorithm configuration and automated algorithm selection in solving combinatorial optimization problems, in this paper, we investigate the efficacy of automated algorithm configuration and automated algorithm selection for the CCAG problem, and propose a novel, automated CCAG approach called *AutoCCAG*. Extensive experiments on public benchmarks show that *AutoCCAG* can find much smaller-sized CCAs than current state-of-the-art approaches, indicating the effectiveness of *AutoCCAG*. More encouragingly, to our best knowledge, our paper reports the first results for CCAG with a high coverage strength (*i.e.,* 5-way CCAG) on public benchmarks. Our results demonstrate that *AutoCCAG* can bring considerable benefits in testing highly configurable software systems.**

***Index Terms*—Constrained Covering Array Generation, Automated Algorithm Optimization**

I. Introduction

Nowadays, there are increasing demands for customized software and services. Developing highly configurable systems has attracted considerable attention in both academia and industry. A highly configurable system provides many options, through which users can easily customize the system [1]–[3]. However, testing such a highly configurable system is challenging. It is hard or even infeasible to test all possible configurations (combinations of options) [4], [5], as the number of configurations grows exponentially with the number of options and only certain specific configurations may lead to system failures. For example, assuming that a software system has 15 options with 3 possible values each, there are more than ten million ($3^{15} = 14,348,907$) possible configurations in the worst case. Hence, the time required for testing all these configurations could be unacceptably high, which creates an urgent need for more practical testing methods.

It is well recognized that combinatorial interaction testing (CIT) [4]–[19] is an effective and practical way for detecting option-combination related faults in a configurable software system. More generally, CIT tests a moderate number of configurations sampled from the entire configuration space [18], thereby significantly reducing the number of required test cases. This sampling process for CIT generates a covering array (CA). A t-way CA covers all possible combinations of the values of any set of t configuration options, where t, the coverage strength, is a small integer value (usually ranging from 2 to 5) [5], [18]. In practice, for configurable systems, there are also hard constraints (mutual dependencies and exclusiveness) among the options. The problem of constrained covering array generation (CCAG), which aims to find minimum-sized constrained covering arrays (CCAs) while satisfying a given set of hard constraints, is the core problem of CIT and is in theory NP-hard [20].

Practical algorithms for tackling the CCAG problem can be categorized into three main classes: greedy algorithms (*e.g.,* [6], [7], [11], [12], [16]), constraint-encoding algorithms (*e.g.,* [9], [13]), and meta-heuristic algorithms (*e.g.,* [8], [10], [14], [15], [17], [18]). Greedy algorithms can handle 2-way and 3-way CCAG rapidly, but the CCAs produced by them are often very large and can be unacceptable in practical scenarios where considerable time is required for testing a single configuration [8], [14]. Constraint-encoding algorithms first encode a given CCAG instance into a constraint optimization problem and then solve it using a constraint solver. Although constraint-encoding algorithms can solve 2-way CCAG, they typically fail to tackle 3-way CCAG. In contrast, meta-heuristic algorithms are able to tackle both 2-way and 3-way CCAG, and usually produce much smaller-sized 2-way and 3-way CCAs than the greedy algorithms. However, effectively solving t-way CCAG ($t \geqslant 4$) still remains a challenge for meta-heuristic algorithms [18], [21].

It is important to solve t-way CCAG with $t \geqslant 4$. Much work (*e.g.,* [4], [19], [22]–[24]) has provided evidence that higher

Qingwei Lin is the corresponding author of this work.

coverage strength indicates stronger fault detection capability. The literature [22] shows that 3-way CCAG detects only 74% of faults for the widely-used traffic collision avoidance system [25]–[27], while 4-way and 5-way CCAG can detect 89% and 100% of faults for that system, respectively. Also, a recent work [23] shows that up to about 25% of faults would be missed if we do not use combinatorial testing with high coverage strength ($t \geqslant 4$). Another recent work [24] demonstrates that, through extensive empirical study on various software, the fault detection rate of t-way CCAG with $t \geqslant 4$ is up to 9.54% more than that of 3-way CCAG. Actually, for life-critical applications (*e.g.,* aviation), even one fault can be fatal [4], [23]. Furthermore, t-way CCAG with $t \geqslant 4$ can find corner-case faults that could cause serious consequences and are very difficult to be detected through manual testing. For example, a recent study in LG Electronics [19] shows that, through 4-way and 5-way combinatorial testing, LG Electronics detected critical faults in washing machines and refrigerators. More importantly, these faults could cost tens of millions of dollars if they had to be fixed after sale [19].

To our best knowledge, very few work focuses on effective solving of the challenging 4-way and 5-way CCAG problems. For example, recently a GPU-enabled parallel CCAG algorithm called *CHiP* [18] reports the experimental results for 4-way CCAG on a number of benchmarking instances, but the CCA sizes reported by *CHiP* are remarkably large and the runtime of *CHiP* is considerably long (which can be observed in Section V and in the literature [18]).

In this paper, for the first time, we study the efficacy of automated algorithm optimization techniques for CCAG, with a focus on solving t-way CCAG with $t = 4$ and $t = 5$. Many meta-heuristic algorithms expose parameters whose settings greatly affect their performance [28]. For instance, in the context of CIT, a representative, state-of-the-art meta-heuristic algorithm called *TCA* [14], which is able to produce small CCAs in many cases [14], [16], introduces a number of parameters that have substantial impact on its performance. Automated algorithm optimization techniques, including automated algorithm configuration (*e.g.,* [28]–[34]) and automated algorithm selection (*e.g.,* [35]–[40]), have been demonstrated to be effective on a variety of NP-hard combinatorial problems such as Boolean satisfiability (*e.g.,* [28], [31], [35]) and minimum vertex cover (*e.g.,* [41]). In this paper, we conduct research on leveraging automated algorithm optimization techniques to optimize the existing CCAG algorithm *TCA*, in order to make it more capable of solving 4-way and 5-way CCAG.

Specifically, we propose a novel, automated CCAG approach, dubbed *AutoCCAG*, which can automatically schedule *TCA* with different configurations for solving t-way CCAG effectively, through leveraging the automated algorithm configuration and automated algorithm selection techniques.

Through extensive experiments, we present that *AutoCCAG* achieves much better performance than current state-of-the-art CCAG algorithms, including *TCA* [14], *CASA* [8] and *CHiP* [18]. In particular, our comparative experiments are conducted on a broad range of real-world application instances from public benchmarks. Our experimental results clearly demonstrate that *AutoCCAG* significantly outperforms current state-of-the-art CCAG algorithms for 4-way and 5-way CCAG on real-world instances. In addition, the performance of *AutoCCAG* is better than or equal to that of all its state-of-the-art CCAG competitors for 2-way and 3-way CCAG on all these instances.

Our main contributions in this paper are as follows.

- We provide clear empirical evidences that automated algorithm configuration and automated algorithm selection can push forward the state of the art in CCAG solving.
- We propose a novel, automated CCAG approach dubbed *AutoCCAG*, which leverages the effectiveness of automated configuration and automated selection techniques.
- We perform extensive experiments, demonstrating that *AutoCCAG* performs significantly better than existing state-of-the-art CCAG algorithms for solving 4-way and 5-way CCAG on real-world instances.

II. Preliminaries

In this section, we first introduce combinatorial interaction testing, and then survey automated algorithm optimization.

A. Combinatorial Interaction Testing

We introduce definitions and notations related to CIT and formally describe the CCAG problem.

a) System Under Test (SUT): The definition of a system under test (SUT, also known as instance in this paper) is a pair $S = (P, C)$, where P is a collection of options and C is a collection of constraints on the permissible combinations of values of the options in P. For each option $p_i \in P$, the set of feasible values is denoted as V_i.

To formally define the CCAG problem, we need to introduce the definitions of *tuple* and *test case*, as described below.

b) Tuple: Given an SUT $S = (P, C)$, a tuple is a set of pairs, denoted by $\tau = \{(p_{i_1}, v_{i_1}), (p_{i_2}, v_{i_2}), \ldots, (p_{i_t}, v_{i_t})\}$, which implies that option $p_{i_j} \in P$ takes the value $v_{i_j} \in V_{i_j}$. A tuple of size t is called a t-tuple.

c) Test Case: Given an SUT $S = (P, C)$, a test case tc is a tuple that covers all options in P. In another word, a test case is a complete assignment to P.

In practice, the options of most software systems are subject to hard constraints on the allowable combination of values. Since testing with invalid test cases would waste much testing time, it is critical to guarantee that all generated test cases are valid. Given an SUT $S = (P, C)$, a tuple or test case is *valid* if, and only if, it does not violate any constraint in C. Besides, a tuple τ is *covered by test case* tc if, and only if, $\tau \subseteq tc$, that is, the options in τ take the same values as the ones in tc.

Since all necessary notations are defined, we introduce the concept of constrained covering array (CCA) and the formal formulation of the constrained covering array generation (CCAG) problem as below.

d) Constrained Covering Array (CCA): Given an SUT $S = (P, C)$, a *t-way constrained covering array* $CCA(S, t)$ is a set of valid test cases, such that any valid t-tuple is covered by at least one test case in CCA, where t is called the *covering strength* of CCA.

e) Constrained Covering Array Generation (CCAG): Given an SUT $S = (P, C)$ and a covering strength t, the problem of t-way constrained covering array generation (CCAG) is to find a t-way CCA of minimum size.

In practice, meta-heuristic algorithms [8], [10], [14], [15], [17], [18] can construct much smaller-sized CCAs than other types of algorithms. Among existing meta-heuristic algorithms for CCAG, *TCA* [14] is considered as the representative and state-of-the-art one. *TCA* is a typical local search CCAG algorithm, which starts from a (partial) CCA as its initial solution, and iteratively improves the current solution via making small modifications. Reported by the literature [14], *TCA* can produce CCAs with much smaller sizes than existing approaches on extensive 2-way and 3-way CCAG instances.

B. Automated Algorithm Optimization Techniques

We first describe automated algorithm configuration, and then introduce automated algorithm selection.

1) Automated Algorithm Configuration: Actually, many practical algorithms have hyper-parameters whose settings considerably affect performance; this especially holds for meta-heuristic algorithms for solving combinatorial optimization problems [28]. The *automated algorithm configuration* (also known as *automated hyper-parameter tuning*) technique is to address the following question: given a configurable algorithm and a set of instances, how to determine the optimized *configuration* (also known as *hyper-parameter settings*) of the given algorithm for solving the given set of instances? Recently, there has been a growing body of automated algorithm configuration for determining optimized hyper-parameter settings [28]–[34]. Automated configuration has been successful applied in various fields, such as data mining [42], automated machine learning [43], and deep learning [44].

2) Automated Algorithm Selection: The automated algorithm selection technique is to address the following question: when there exist a number of base algorithms aiming at solving the identical problem, how to select the most suitable one? Considerable attentions have been paid to this question, resulting in various promising approaches [35]–[39]. Automated algorithm selection has shown its effectiveness in solving many NP-hard combinatorial problems, such as Boolean satisfiability [35], maximum satisfiability [45], answer set programming [46], and constraint satisfaction problem [39].

III. The *AutoCCAG* Approach

In this section, we propose a novel, effective automated CCAG approach called *AutoCCAG*, which leverages effective automated configuration and automated selection techniques.

A. Top-level Design of AutoCCAG

The main idea of our proposed *AutoCCAG* approach is to advance the state of the art in CCAG solving through automated algorithm configuration and automated algorithm selection. We first illustrate the top-level design of *AutoCCAG* in Figure 1. There are three key components in our *AutoCCAG* approach: 1) configuration optimizer; 2) promising configuration generator; 3) configuration scheduling planner.

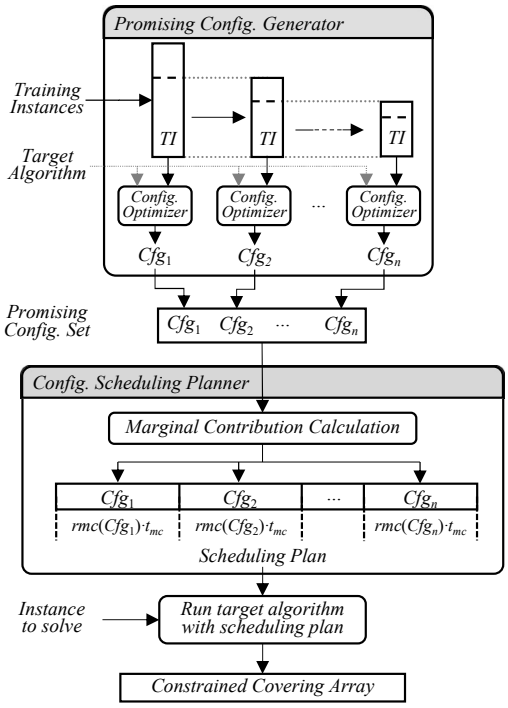

Fig. 1. Top-level design of *AutoCCAG*.

1) Workflow of AutoCCAG*:* As illustrated in Figure 1, *AutoCCAG* works as follows: Given a target, configurable CCAG algorithm a, the configuration optimizer of *AutoCCAG* utilizes automated configuration to determine the optimized configuration for a; the promising configuration generator of *AutoCCAG* aims to generate a set of promising configurations of a with complementary strength; the configuration scheduling planner of *AutoCCAG* leverages automated selection to construct a scheduling plan for target CCAG algorithm a. Finally, *AutoCCAG* runs target CCAG algorithm a with the generated scheduling plan to solve a given instance, to produce the constrained covering array for that instance.

2) Target CCAG Algorithm: As discussed before, we first need to decide the target CCAG algorithm adopted by *AutoCCAG*. As reported in the literature [14] and also observed in our experiments (in Tables I and II), *TCA* [14] can produce notably smaller-sized CCAs than other existing algorithms on t-way CCAG ($2 \leqslant t \leqslant 5$). Also, *TCA* is a configurable algorithm, and its original version has 3 configurable hyper-parameters. One hyper-parameter is Boolean-valued, and has two possible values: *True* or *False*. Another one is a positive integer hyper-parameter. The remaining one is a real-valued hyper-parameter ranging from 0 to 1. In fact, a powerful paradigm called programming by optimization (PbO) [47], which advocates practitioners to expand the design space of target algorithms, has shown its effectiveness in improving meta-heuristic algorithms for solving a variety of NP-hard problems, *e.g.,* Boolean satisfiability [48] and minimum vertex cover [41]. Inspired by the success of the PbO paradigm, we expand the design space of *TCA* through the PbO paradigm,

Alg. 1: Method *BOAC* for Configuration Optimizer

Input: $\boldsymbol{TB\_BOAC}$: time budget for *BOAC*;
$\boldsymbol{TI}$: a collection of training instances;
$\boldsymbol{a}$: the target algorithm;
Output: $\boldsymbol{Cfg^*}$: optimized configuration of a;

1 $Cfg \leftarrow$ the default configuration of a;
2 $Perf \leftarrow$ the average performance of a with Cfg on TI;
3 $(Cfg^*, Perf^*) \leftarrow (Cfg, Perf)$;
4 $ML \leftarrow$ a GP model trained using sample $(Cfg, Perf)$;
5 **while** *time budget* TB_BOAC *is not reached* **do**
6 $SC \leftarrow$ a set of randomly sampled configurations of a;
7 **if** *with a probability of half* **then**
8 $Cfg \leftarrow$ the configuration of a with the largest *expected improvement* assessed by ML from SC;
9 **else**
10 $Cfg \leftarrow$ the configuration of a with the largest *variance* assessed by ML from SC;
11 $Perf \leftarrow$ the average performance of a with Cfg on TI;
12 **if** $Perf$ *is better than* $Perf^*$ **then** $(Cfg^*, Perf^*) \leftarrow (Cfg, Perf)$;
13 ML is incrementally updated by adding a new sample $(Cfg, Perf)$;
14 **return** Cfg^*

and thus make *TCA* incorporate more configurable algorithm mechanisms. For each of these newly incorporated, configurable algorithm mechanisms, one new, Boolean-valued hyper-parameter is introduced in *TCA* to decide whether this new algorithm mechanism is activated. Hence, *AutoCCAG* utilizes the PbO-based version of *TCA* as its target CCAG algorithm.

B. Configuration Optimizer

Since automatically configured algorithms have exhibited state-of-the-art performance on a wide range of combinatorial problems [28], [31], [49], it is advisable to design a configuration optimizer based on automated algorithm configuration in *AutoCCAG*. Hence, the first step is to investigate to what degree automated algorithm configuration is effective for CCAG.

1) Details of BOAC*:* Bayesian optimization (BO) [50] is an effective framework for automatically tuning hyper-parameters of configurable algorithms [31]. The BO framework constructs and updates a machine learning model to learn the effect of hyper-parameter settings on target algorithm performance, and iteratively determines a promising configuration via the constructed machine learning model. An effective machine learning model for BO is Gaussian process (GP) [51]. Given a configuration of the target, configurable algorithm, GP can evaluate its potential benefit using *expected improvement* (EI) [52] and assess its diversification property using *variance* [53].

Based on the BO framework, we propose a new method called *BOAC* (*Bayesian Optimization based Automated Configuration*) for our configuration optimizer. *BOAC* utilizes GP as its machine learning model. The *BOAC* method is outlined in Alg. 1, and needs 3 inputs: 1) the time budget for *BOAC*, denoted by TB_BOAC; 2) a collection of training instances, denoted by TI; 3) the target algorithm to be configured, denoted by a. The output of *BOAC* is the optimized configuration of the target algorithm a, denoted by Cfg^*.

BOAC works in an iterative manner until the time budget TB_BOAC for *BOAC* is reached (Line 5 in Alg. 1). In each iteration, *BOAC* obtains a new sample, *i.e.,* a pairwise item consisting of a configuration chosen by GP, denoted by Cfg, and the average performance of a with Cfg across all training instances in TI, denoted by $Perf$ (Lines 8, 10 and 11 in Alg. 1)[1]. After obtaining the new sample, in each iteration *BOAC*'s machine learning model ML would be incrementally updated by adding a new sample $(Cfg, Perf)$ (Line 13 in Alg. 1). Then we need to specify how *BOAC* chooses a promising configuration using GP in each iteration. In each iteration, *BOAC* first constructs a candidate configuration set SC, where each candidate is randomly sampled from the whole configuration space (Line 6 in Alg. 1). Then *BOAC* switches between the exploitation mode and the exploration mode to pick a promising configuration. Since it is important to balance exploitation and exploration [31], in our work, with a probability of half, *BOAC* works in the exploitation mode: it picks the configuration Cfg with the largest EI assessed by ML from SC (Line 8 in Alg. 1); otherwise, *BOAC* works in the exploration mode: it chooses the configuration Cfg with the largest *variance* assessed by ML from SC (Line 10 in Alg. 1). Since in each iteration *BOAC*'s machine learning model ML would be incrementally updated by using a new sample, the configuration determined by *BOAC* would become more effective with the number of iterations increases [54].

2) Configuration Protocol: In this paper, *AutoCCAG* applies our proposed algorithm optimization method *BOAC* to conduct automated algorithm configuration. For automated configuration, we use one benchmark as the training set. Our *BOAC* method is utilized to minimize the size of the generated CCA. Following the common setting of automated configuration [55], we use a time budget of 2 days for the configuration process of *BOAC*, and a cutoff time of 300 seconds per algorithm run during the configuration process of *BOAC*. Once the configuration process is completed, the configuration determined by *BOAC* is reported as the final outcome of our configuration process.

C. Promising Configuration Generator

In Section III-B, we introduce how to use configuration optimizer to determine the optimized configuration for an configurable algorithm on a given collection of training instances. We note that the obtained optimized configuration shows the best average performance across all training instances; however, this does not mean that the obtained optimized configuration can achieve the best performance on each training instance. In fact, it is recognized that, for the same configurable algorithm, its various configurations show diverse performance when solving different instances [56]. Hence, it is advisable to generate a set of promising configurations with complementary strength, and then use the automated selection technique to leverage the complementary strength of them.

[1]For the first iteration, since the GP model ML has not been constructed, the default configuration of a and the average performance of a with the default configuration are chosen (Lines 1 and 2 in Alg. 1).

Alg. 2: Method *PCG* for Promising Configuration Generator

Input: $\boldsymbol{IB\_PCG}$: iteration budget for *PCG*;
$\boldsymbol{TB\_BOAC}$: time budget for *BOAC*;
$\boldsymbol{TI}$: a set of training instances;
$\boldsymbol{a}$: the target algorithm;
Output: $\boldsymbol{PC}$: a set of promising configurations of a;

1 $PC \leftarrow \varnothing$;
2 **while** *time budget* IB_PCG *is not reached* and $TI \neq \varnothing$ **do**
3 $Cfg^* \leftarrow$ the optimized configuration of target algorithm a selected by *BOAC* using TB_BOAC, TI as inputs;
4 $PC \leftarrow PC \cup \{Cfg^*\}$;
5 remove such instances, where target algorithm a with configuration Cfg^* can achieve or exceed best known CCA sizes, from TI;
6 **return** PC

To address this challenge, we propose a new method dubbed *PCG*, which is a promising configuration generator to recommend a set of promising configurations with complementary strength. Our proposed *PCG* method is listed in Alg. 2, and needs 4 inputs: 1) IB_PCG, *i.e.,* the iteration budget for *PCG* (following the practical standard [57], in this paper IB_PCG is set to 4); 2) TB_BOAC, *i.e.,* the time budget for *BOAC*; 3) TI, *i.e.,* a collection of training instances; 4) a, *i.e.,* the target algorithm. The output of *PCG* is a set of promising configurations, denoted by PC.

In the beginning, *PCG* initializes PC as an empty set (Line 1 in Alg. 2). Then *PCG* works in an iterative manner until one of the termination criteria is met. As indicated in Alg. 2, there are two termination criteria for *PCG*: 1) the iteration budget IB_PCG is reached; 2) the training instance set TI becomes empty. In each iteration, *PCG* first activates *BOAC* to determine the optimized configuration Cfg^* using TB_BOAC, TI and a as inputs (Line 3 in Alg. 2); then *PCG* adds Cfg^* into PC (Line 4 in Alg. 2). At the end of each iteration, *PCG* removes such instances, where target algorithm a with the configuration Cfg^* found in this iteration can achieve or exceed best known CCA sizes[2], from TI (Line 5 in Alg. 2); the main intuition is that, in the subsequent iterations we only focus on those instances where the configurations already in PC show moderate performance, in order to better find those configurations with complementary strength.

In summary, the main idea of *PCG* is to find such configurations, which can achieve better performance on those instances where previously determined configurations show moderate performance. As a result, *PCG* is able to generate a set of promising configurations with complementary strength on all training instances.

D. Configuration Scheduling Planner

Thanks to our proposed *PCG* method, for a given CCAG algorithm a (*i.e., TCA* in this paper), we can obtain a set of promising configurations PC with complementary strength on all training instances.

[2]For the best known CCA size for each instance, we collect the value from previous studies [8], [14], [15], [18].

It has been widely recognized that providing a *scheduling plan* consisting of a combination of different effective configurations is able to achieve significantly better results than just determining a single configuration [37]. The intuition is that running a scheduling plan would exploit the complementary strength among all configurations in this plan, and could achieve more robust performance than just using a single configuration [37]. Also, the literature [58] conducts extensive empirical study to demonstrate that, for the Boolean satisfiability (SAT) problem, which is a well-known, prototypical NP-hard combinatorial problem, the effective configurations for solving the SAT problem are usually not similar, resulting in different clusters of effective configurations in the entire configuration space. Besides, the literature [9], [13] demonstrates that there is a strong connection between the CCAG problem and the SAT problem (the CCAG problem can be encoded into the SAT problem), so it is possible that there might be different clusters of effective configurations for solving the CCAG problem. Hence, an advisable solution is to provide a scheduling plan which consists of a bunch of configurations with different time budgets rather than a single configuration with the whole time budget. Then we need to address the following challenge: given a new instance and the cutoff time for solving the new instance, how to select suitable configurations from PC and allocate suitable time budget for each selected configuration?

In order to address this challenge, we propose a novel method named *CSP*, which is a configuration scheduling planner based on the automated selection technique.

1) Marginal Contribution: Before introducing the details of our *CSP* method, we introduce an important concept called *marginal contribution* [59], which can measure the contribution of each configuration underlying the whole scheduling plan. Since the methods for computing marginal contribution for various problems are different, we utilize a logarithmic-based method for computing marginal contribution in the context of CCAG solving, which is described as follows.

Given a set of *TCA*'s promising configurations PC, notation $size(PC)$ denotes the performance (*i.e.,* the averaged size of resulting CCAs on all training instances) achieved by an ideal configuration selector which leverages the complementary strengths of the TCA's configurations in PC. The *absolute marginal contribution* (amc) for each configuration $b \in PC$ is calculated below:

$$amc(b) = \ln \frac{size(PC \backslash \{b\})}{size(PC)} \tag{1}$$

After obtaining the absolute marginal contribution for each configuration $b \in PC$, the *relative marginal contribution* (rmc) for each configuration $b \in PC$ is calculated below:

$$rmc(b) = \frac{amc(b)}{\sum_{c \in PC} amc(c)} \tag{2}$$

2) Details of CSP*:* Our *CSP* method is outlined in Alg. 3, and provides a scheduling plan based on the marginal contribution calculation. The time budget assigned to our *CSP* method is the whole given cutoff time.

Alg. 3: Method *CSP* for Configuration Scheduling Planner

Input: t_{mc}: cutoff time for solving instance i;
PC: a set of promising configurations of the target algorithm found by *SPC*;
Output: $list_{mc}$: optimized configuration schedule plan;

```
list_tmp ← [ ];
foreach configuration b ∈ PC do
    calculate rmc(b) according to Equation 2;
    list_tmp.append([b, rmc(b)]);
sort list_tmp by rmc in a descending order;
list_mc ← [ ];
foreach pairwise tuple [b, rmc(b)] in list_tmp do
    list_mc.append([b, rmc(b) · t_mc]);
return list_mc;
```

As described in Alg. 3, the procedures of our *CSP* method are described as follows: For each configuration b underlying the given set of promising configurations PC, the relative marginal contribution for configuration b (*i.e.,* $rmc(b)$) is computed according to Equation 2. Then all configurations are sorted by their rmc values in a descending order. The scheduling plan generated in this stage consists of all configurations whose rmc values are greater than 0, and the time budget allocated to each component algorithm is proportional to its rmc value.

Finally, through the above procedures, *CSP* constructs the final scheduling plan $list_{mc}$. After the final scheduling plan $list_{mc}$ is obtained, our *AutoCCAG* approach runs the target CCAG algorithm with the final scheduling plan $list_{mc}$ to solve a given instance, and thus generates the constrained covering array for that instance.

IV. Experimental Design

In this section, we describe the experimental design of this work in detail. In particular, we first introduce the benchmarks used in the experiments. Then we present the research questions of this paper. After that, we describe the competitors. Finally, we introduce the experimental setup.

A. Benchmarks

Since the literature [14] utilizes two benchmarks (*i.e.,* `Real-world` and `Synthetic`) to evaluate the performance of *TCA*, we therefore adopt these two benchmarks in our experiments. Both benchmarks are readily available online. Moreover, we include an additional real-world application benchmark entitled `IBM` into our experiments.

In our experiment, a benchmark is a collection of CCAG instances, where each CCAG instance consists of two files, *i.e.,* the model file and the constraint file. More details about benchmarks are available online[3]. We briefly describe the `Real-world`, `IBM` and `Synthetic` benchmarks below.

`Real-world`. This benchmark includes 5 real-world instances and has been intensely studied in literature [8], [10], [14], [15], [18], [22], [60], [61]. These instances are derived from Apache, an influential open-source web sever application; Bugzilla, a widely used web-based bug tracker; GCC, a well-known compiler collection from the GNU community containing compilers and libraries for multiple programming languages; SPIN-S and SPIN-V, the simulator- and verifier-variants of SPIN, a widely-used model checking tool. This benchmark was first presented by Cohen *et al.* [60][4].

`IBM`. This real application benchmark is comprised of 20 practical instances, originally introduced in the literature [62], and is available online[5]. These instances are generated aiming to provider better service for IBM customers, and cover a broad range of real-world application fields, including health care, insurance, network management, storage, *etc*.

`Synthetic`. This benchmark contains 30 synthetic instances that were generated to resemble the characteristics of real-world software systems in `Real-world`. These synthetic instances were originally described by Cohen *et al.* [61][4].

The `Synthetic` benchmark is used as the training set required by *AutoCCAG*. In this paper, *AutoCCAG* is trained by solving 3-way CCAG on all instances in the `Synthetic` benchmark. The `Real-world` and `IBM` benchmarks are adopted as the testing set and are used to evaluate the practical performance of *AutoCCAG* and other state-of-the-art CCAG algorithms. Due to the page limit, we only list the results of *AutoCCAG* and its competitors for solving 4-way and 5-way CCAG on `Real-world` and 10 hardest instances in `IBM` in Tables I–IV. The complete results of *AutoCCAG* and its competitors for solving 4-way and 5-way CCAG on all instances in the `IBM` benchmark are available online[3] (where the results on all instances in the `Real-world` and `IBM` benchmarks for 2-way and 3-way CCAG can be also found).

B. Research Questions

To evaluate the effectiveness of *AutoCCAG*, we aim to answer the following research questions (RQs). In the context of CCAG solving, previous meta-heuristic solvers can achieve good performance for 2-way and 3-way CCAG, but effectively solving 4-way and 5-way CCAG still remains a challenge [18], [21]. Hence, in this paper, we focus on advancing the current state of the art in 4-way and 5-way CCAG solving.

RQ1: Can the use of automated configuration improve the state of the art in 4-way and 5-way CCAG solving?

In this RQ, we evaluate the efficacy of our proposed configuration optimizer *BOAC* for 4-way and 5-way CCAG. We would like to explore if the performance of the state-of-the-art CCAG algorithm *TCA* can be improved through *BOAC*. In particular, we first utilize *BOAC* to find the optimized configuration for the original version of *TCA*, resulting in the original version of *TCA* with the optimized configuration, dubbed *TCA-opt*. Then we conduct experiments to demonstrate whether *TCA-opt* shows performance improvement over *TCA*.

RQ2: Can automated selection be leveraged to improve the state of the art in 4-way and 5-way CCAG solving?

[3]https://github.com/chuanluocs/AutoCCAG

[4]http://cse.unl.edu/~citportal/public/tools/casa/benchmarks.zip

[5]https://researcher.watson.ibm.com/researcher/files/il-ITAIS/ctdBenchmarks.zip

TABLE I
COMPARING *AutoCCAG* WITH *TCA-opt*, *TCA*, *CASA* AND *CHiP* FOR 4-WAY CCAG ON THE Real-world AND IBM BENCHMARKS. THE RUN TIME IS MEASURED IN SECOND.

Instance	*AutoCCAG*		*TCA-opt*		*TCA*		*CASA*		*CHiP*	
	min (avg)	time	min (avg)	time	min (avg)	time	min (avg)	time	size	time
Apache	**772 (789.2)**	8068.3	838 (838.0)	9652.3	– (–)	>10000	– (–)	>10000	838	86169
Bugzilla	**167 (167.8)**	428.4	168 (169.6)	637.6	171 (172.4)	896.7	274 (280.7)	931.0	176	20821
GCC	**374 (379.2)**	8477.7	444 (444.0)	8199.0	– (–)	>10000	– (–)	>10000	444	103177
SPIN-S	**308 (308.0)**	44.1	**308 (308.0)**	121.6	311 (317.6)	331.7	355 (362.1)	862.6	339	12853
SPIN-V	**1113 (1117.0)**	911.5	1562 (1562.6)	937.6	1637 (1655.3)	981.9	– (–)	>1000	1166	33773
Healthcare2	**159 (166.6)**	757.7	169 (171.4)	376.6	171 (173.3)	463.1	184 (186.8)	298.6	177	905
Healthcare3	**723 (729.1)**	768.1	757 (763.4)	851.6	770 (773.1)	928.0	1127 (1159.4)	955.3	851	29062
Healthcare4	**1317 (1320.4)**	906.7	1475 (1494.3)	987.1	1763 (1791.2)	997.2	2492 (2605.8)	944.0	1536	36719
Insurance	**75361 (75361.0)**	165.4	75486 (75489.1)	992.1	76273 (76310.5)	997.4	474131 (613700.6)	972.0	75764	213365
NetworkMgmt	**5610 (5610.0)**	233.8	**5610 (5610.0)**	337.7	**5610** (5610.1)	561.1	6008 (6045.9)	975.5	**5610**	207136
ProcessorComm1	**485 (487.8)**	452.2	489 (491.7)	461.2	491 (495.3)	633.2	571 (589.1)	902.0	544	11187
ProcessorComm2	**574 (575.1)**	595.7	585 (587.0)	517.3	592 (595.6)	876.5	850 (869.9)	961.3	837	7942
Services	**6404 (6407.8)**	807.7	6409 (6414.1)	853.4	6419 (6431.3)	961.1	7198 (7312.1)	953.7	6855	441412
Storage4	**5486 (5494.0)**	951.9	6077 (6084.0)	993.8	6774 (6805.1)	999.0	9393 (9500.5)	942.1	5671	86073
Storage5	**10982 (11010.8)**	821.8	13161 (13163.1)	971.0	14067 (14104.0)	993.1	– (–)	>1000	13292	36930

In this RQ, to study the effectiveness of automated selection, we conduct comparisons between *AutoCCAG* and *TCA-opt*. Actually, *TCA-opt*, which runs the original version of *TCA* with a configuration selected by *BOAC*, does not leverage the effectiveness of automated selection.

RQ3: How does *AutoCCAG* compare against state-of-the-art algorithms for 4-way and 5-way CCAG?

In this RQ, *AutoCCAG* is compared against three state-of-the-art CCAG algorithms, *i.e., TCA* [14], *CASA* [8] and *CHiP* [18], for solving 4-way and 5-way CCAG.

RQ4: Can *AutoCCAG* show state-of-the-art performance for 4-way and 5-way CCAG with a shorter cutoff time?

In this RQ, *AutoCCAG* is evaluated to solve 4-way and 5-way CCAG instances with a half of the cutoff time. We compare the results of *AutoCCAG* using a half of the cutoff time against the results of *TCA-opt* using the full cutoff time, to demonstrate the efficiency of *AutoCCAG*.

C. Competitors

In this paper, *AutoCCAG* is compared with 3 state-of-the-art CCAG algorithms, *i.e., TCA* [14], *CASA* [8] and *CHiP* [18].

TCA [14] is a state-of-the-art two-mode meta-heuristic algorithm. Reported in the literature [14], *TCA* outperforms a number of CCAG algorithms including *CASA* [8], *Cascade* [12] and *ACTS* [11] on many real-world and synthetic instances.

CASA [8] is a high-performance simulated annealing CCAG algorithm[6]. Reported in the literature [8], *CASA* outperforms a greedy construction algorithm *mAETG* [61] on a number of real-world and synthetic instances.

CHiP [18] is a recently-proposed, effective hybrid parallel CCAG algorithm, which can use vast amount of parallelism provided by graphics processing units. As reported in the literature [18] and claimed by the authors of *CHiP* [18], *CHiP* reports the first and state-of-the-art results for solving 4-way CCAG on the Real-world and IBM benchmarks.

[6]http://cse.unl.edu/~citportal/

We note that *HHSA* [15] is an effective CCAG algorithm for 2-way and 3-way CCAG. Since the implementation of *HHSA* available online[7] does not support solving t-way CCAG with $t \geqslant 4$, we do not include *HHSA* into our comparisons for 4-way and 5-way CCAG. The results of comparing *AutoCCAG* with *HHSA* for 2-way and 3-way CCAG are available online[3], and *AutoCCAG* can find smaller-sized or equal-sized CCAs than *HHSA* on all instances for 2-way and 3-way CCAG.

D. Experimental Setup

All experiments in this paper were conducted on a computing server with 2.50GHz Intel Xeon E7-8890 v3 CPU and 1.0TB memory, running GNU/Linux. Because meta-heuristic algorithms are usually randomized, we performed 10 independent runs per instance for each algorithm. For solving 4-way CCAG, the cutoff time of each algorithm run is set to 1,000 CPU seconds, following the experimental setup of *TCA* [14]. Since no CCAG algorithm can report feasible solutions for 4-way CCAG on the 'Apache' and 'GCC' instances within 1,000 CPU seconds, in our experiments the cutoff time of each algorithm run for solving 4-way CCAG on the 'Apache' and 'GCC' instances is set to 10,000 CPU seconds. It is recognized that solving 5-way CCAG is much more difficult than solving 4-way CCAG, and solving 5-way CCAG requires vast computing time [21]. Thus, for 5-way CCAG, the cutoff time of each algorithm run is set to 10,000 CPU seconds.

In our experiments (especially in Tables I–IV), for *TCA*, if there is no specific description, it is evaluated using the original version with the default configuration. For *CASA*, it is evaluated using the configuration recommended by its authors [8]. For *CHiP*, we do not have the access to its source code, and only its binary executable is available[8]. We tried to run the binary executable of *CHiP* on all instances for solving 4-way CCAG using the cutoff time of 1,000 seconds (the same

[7]http://www0.cs.ucl.ac.uk/staff/Y.Jia/projects/cit_hyperheuristic/downloads/Comb_Linux_64.tar.gz

[8]https://github.com/susoftgroup/CHiP

TABLE II
COMPARING *AutoCCAG* WITH *TCA-opt*, *TCA*, *CASA* AND *CHiP* FOR 5-WAY CCAG ON THE REAL-WORLD AND IBM BENCHMARKS. THE RUN TIME IS MEASURED IN SECOND.

Instance	*AutoCCAG*		*TCA-opt*		*TCA*		*CASA*		*CHiP*	
	min (avg)	time	min (avg)	time	min (avg)	time	min (avg)	time	size	time
Apache	– (–)	>10000	– (–)	>10000	– (–)	>10000	– (–)	>10000	–	–
Bugzilla	**560 (561.5)**	8200.2	688 (688.9)	9665.4	788 (794.5)	9922.2	1194 (1480.7)	7787.6	–	–
GCC	– (–)	>10000	– (–)	>10000	– (–)	>10000	– (–)	>10000	–	–
SPIN-S	**1174 (1174.0)**	860.2	**1174 (1174.0)**	2908.4	**1174 (1174.0)**	2987.9	1222 (1228.8)	9589.3	–	–
SPIN-V	**5941 (6060.4)**	8991.6	8202 (8202.0)	3376.0	– (–)	>10000	– (–)	>10000	–	–
Healthcare2	**517 (520.9)**	6926.7	**517** (521.0)	6422.9	521 (523.7)	6352.7	558 (574.2)	4524.8	–	–
Healthcare3	**3197 (3207.5)**	8229.9	3934 (3938.2)	9944.8	4214 (4243.6)	9973.2	9908 (12729.9)	9572.7	–	–
Healthcare4	**6885 (6904.5)**	9594.7	8184 (8184.0)	9527.0	9353 (9380.3)	9624.7	57417 (57417.6)	6155.1	–	–
Insurance	**452575 (452779.8)**	9912.4	491558 (491561.7)	9900.7	496934 (497147.8)	9946.2	– (–)	>10000	–	–
NetworkMgmt	**24664 (24679.9)**	9875.5	24665 (24680.2)	9874.4	24705 (24721.2)	9915.8	28209 (28382.9)	9737.4	–	–
ProcessorComm1	**2037 (2038.7)**	5987.4	2041 (2043.6)	7643.3	2042 (2044.7)	8157.0	2586 (2630.7)	9829.3	–	–
ProcessorComm2	**2506 (2508.9)**	5938.1	2583 (2588.3)	9600.1	2808 (2886.8)	9990.1	4293 (4895.8)	9620.5	–	–
Services	**32869 (32887.9)**	9315.2	33208 (33243.9)	9890.9	36510 (36616.9)	9998.0	42928 (43319.7)	9592.2	–	–
Storage4	**34005 (34027.1)**	9882.3	39222 (39228.3)	9939.4	43552 (43632.1)	9959.8	192964 (264817.2)	9296.9	–	–
Storage5	**65854 (66047.4)**	9561.7	78317 (78318.2)	9457.1	85304 (85411.2)	9728.6	– (–)	>10000	–	–

cutoff time adopted by *AutoCCAG*), but in our experimental environment *CHiP* cannot report feasible solutions for almost all instances. Actually, this is not surprising; the experimental results in the literature [18] report that, for solving 4-way CCAG, on the majority of instances, *CHiP* requires more than tens of thousands (and even more than hundreds of thousands) of seconds and needs vast amount of parallelism provided by graphics processing units to obtain feasible solutions. As a result, the experimental results of *CHiP* for solving 4-way CCAG are taken from the literature [18]. Also, the binary executable of *CHiP* does not support solving t-way CCAG with $t > 4$, so in Table II we mark '–' for the results of *CHiP* for solving 5-way CCAG.

For each algorithm on each instance, we report the smallest size ('min') and the averaged size ('avg') of the CCA found by the respective algorithm over 10 runs. In addition, for each algorithm on each instance, we report the running time ('time') required for finding the optimized CCAs averaged over 10 runs, and all running times were measured in CPU seconds. If an algorithm failed to find a CCA during all 10 runs, we report the results as '– (–)'. In Tables I–IV, for each instance, we use **boldface** to indicate the best results with regard to CCA size in our comparisons.

Moreover, in Tables I–IV, for each instance, we individually compare the performance of *AutoCCAG* 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 [63] for each pairwise comparison between *AutoCCAG* and each of its competitors. For each instance, if a) all the p-values of Wilcoxon signed-rank tests at 95% confidence level are smaller than 0.05, and b) the Vargha-Delaney effect sizes for all pairwise comparisons (between *AutoCCAG* and each of its competitors) are greater than 0.71 (indicating large effect sizes) [63], [64], we consider the performance improvement of *AutoCCAG* over all its competitors statistically significant and meaningful, and mark the results of *AutoCCAG* using underline. In Table I, the experimental results of *CHiP* are taken directly from the literature [18], which were obtained by only performing one run of *CHiP* per instance. Therefore, we do not conduct statistical test to compare the performance of our *AutoCCAG* approach with that of *CHiP*. For our *BOAC* method, following the literature [65], we adopt the ARD Matérn 5/2 kernel as its GP kernel and use the hyper-parameter settings recommended in the literature [65].

V. EXPERIMENTAL RESULTS

In this section, we report the experimental results, to show both the effectiveness and the efficiency of *AutoCCAG*.

A. RQ1: Performance Improvement for 4-way and 5-way CCAG by Automated Configuration

We utilize *BOAC* to find the optimized configuration for the original version of *TCA*, resulting in *TCA-opt* (*i.e.,* the original version of *TCA* with the optimized configuration). That is to say, for solving a given CCAG instance, *TCA-opt* runs the original version of *TCA* with the single, optimized configuration. The comparative results of *TCA-opt* and *TCA* for 4-way and 5-way CCAG on the `Real-world` and `IBM` benchmarks are reported in Tables I and II, respectively. We note that the full results are available online.[3] As can be clearly seen from Tables I and II, on both metrics of 'smallest size' and 'averaged size', the performance of *TCA-opt* is better than or equal to that of *TCA* on all instances.

The experimental results in Tables I and II demonstrate that the state of the art in 4-way and 5-way CCAG solving can be considerably advanced using automated configuration.

B. RQ2: Performance Improvement for 4-way and 5-way CCAG by Automated Selection

The major difference between *AutoCCAG* and *TCA-opt* is that *AutoCCAG* runs the PbO-based version of *TCA* with a scheduling plan (consisting of multiple high-performing configurations of the PbO-based version of *TCA*), while *TCA-opt* runs the original version of *TCA* with a single, optimized

TABLE III
COMPARING *AutoCCAG* (WITH THE CUTOFF TIME OF 500 SECONDS) AGAINST *TCA-opt* (WITH THE CUTOFF TIME OF 1,000 SECONDS) FOR 4-WAY CCAG ON THE REAL-WORLD AND IBM BENCHMARKS. THE RUN TIME IS MEASURED IN SECOND.

Instance	*AutoCCAG* (500 sec)		*TCA-opt* (1,000 sec)	
	min (avg)	time	min (avg)	time
Apache	– (–)	>500	– (–)	>1000
Bugzilla	**167 (167.9)**	388.2	168 (169.6)	637.6
GCC	– (–)	>500	– (–)	>1000
SPIN-S	**308 (308.0)**	44.1	**308 (308.0)**	121.6
SPIN-V	**1119 (1124.1)**	472.8	1562 (1562.6)	937.6
Healthcare2	**161 (168.0)**	344.3	169 (171.4)	376.6
Healthcare3	**728 (731.8)**	397.2	757 (763.4)	851.6
Healthcare4	**1331 (1335.6)**	476.9	1475 (1494.3)	987.1
Insurance	**75361 (75361.0)**	165.4	75486 (75489.1)	992.1
NetworkMgmt	**5610 (5610.0)**	233.8	**5610 (5610.0)**	337.7
ProcessorComm1	**485 (488.7)**	381.5	489 (491.7)	461.2
ProcessorComm2	**575 (576.5)**	298.7	585 (587.0)	517.3
Services	**6406 (6408.8)**	484.3	6409 (6414.1)	853.4
Storage4	**5518 (5523.5)**	481.3	6077 (6084.0)	993.8
Storage5	**11041 (11053.3)**	452.8	13161 (13163.1)	971.0

TABLE IV
COMPARING *AutoCCAG* (WITH THE CUTOFF TIME OF 5,000 SECONDS) AGAINST *TCA-opt* (WITH THE CUTOFF TIME OF 10,000 SECONDS) FOR 5-WAY CCAG ON THE REAL-WORLD AND IBM BENCHMARKS. THE RUN TIME IS MEASURED IN SECOND.

Instance	*AutoCCAG* (5,000 sec)		*TCA-opt* (10,000 sec)	
	min (avg)	time	min (avg)	time
Apache	– (–)	>5000	– (–)	>10000
Bugzilla	**566 (567.8)**	4531.8	688 (688.9)	9665.4
GCC	– (–)	>5000	– (–)	>10000
SPIN-S	**1174 (1174.0)**	860.2	**1174 (1174.0)**	2908.4
SPIN-V	**6429 (6809.1)**	3231.8	8202 (8202.0)	3376.0
Healthcare2	521 (522.9)	3605.9	**517 (521.0)**	6422.9
Healthcare3	**3210 (3221.5)**	4476.7	3934 (3938.2)	9944.8
Healthcare4	**6973 (6994.0)**	4928.8	8184 (8184.0)	9527.0
Insurance	**456781 (457835.1)**	4999.8	491558 (491561.7)	9900.7
NetworkMgmt	24773 (24788.6)	4963.9	**24665 (24680.2)**	9874.4
ProcessorComm1	**2038 (2040.0)**	4560.3	2041 (2043.6)	7643.3
ProcessorComm2	**2506 (2511.3)**	3582.7	2583 (2588.3)	9600.1
Services	**32987 (33031.0)**	4694.4	33208 (33243.9)	9890.9
Storage4	**34207 (34261.6)**	4966.9	39222 (39228.3)	9939.4
Storage5	**66857 (66966.8)**	4992.9	78317 (78318.2)	9457.1

configuration. Hence, *TCA-opt* does not leverage the effectiveness of automated selection. The comparative results of *AutoCCAG* and *TCA-opt* for 4-way and 5-way CCAG on both Real-world and IBM benchmarks are summarized in Tables I and II, respectively. According to Tables I and II, the results present that, when compared to *TCA-opt*, our *AutoCCAG* approach is able to consistently achieve better or equal performance on all instances in terms of the metrics of 'smallest size' and 'averaged size'.

The experimental results in Tables I and II provide evidence that automated selection can significantly push forward the state of the art in 4-way and 5-way CCAG solving.

C. RQ3: Comparison among AutoCCAG and State-of-the-art CCAG Algorithms for 4-way and 5-way CCAG

Related to this RQ, we conduct experiments on extensive real-world applications instances to compare our *AutoCCAG* approach against existing state-of-the-art CCAG algorithms, in order to show the effectiveness of *AutoCCAG*.

The experimental results of *AutoCCAG* and its state-of-the-art competitors (*i.e., TCA*, *CASA* and *CHiP*) for 4-way and 5-way CCAG on the Real-world and IBM benchmarks are presented in Tables I and II, respectively. It is clear that, *AutoCCAG* performs much better than all its competitors for solving 4-way and 5-way CCAG on these two benchmarks.

For solving 4-way CCAG, on the metric of 'smallest size', *AutoCCAG* performs much better than all its state-of-the-art competitors on 14 out of 15 instances presented in Table I; for the remaining instance (*i.e.,* 'NetworkMgmt'), *AutoCCAG*, *TCA* and *CHiP* can find the CCA with the same smallest size of 5,610 (besides, *CASA* performs worse than *AutoCCAG*, *TCA* and *CHiP* on this instance), but the run time required by *AutoCCAG* (233.8 sec) is much less than *TCA* (561.1 sec) and *CHiP* (207,136 sec). Also, on the metric of 'averaged size', *AutoCCAG* performs much better than all its state-of-the-art competitors on all 15 instances presented in Table I.

For solving 5-way CCAG, except two instances (*i.e.,* 'Apache' and 'GCC') where no CCAG algorithm can report feasible solutions within the cutoff time, on both metrics of 'smallest size' and 'averaged size', *AutoCCAG* achieves much better performance than all its state-of-the-art competitors on 12 out of 13 instances presented in Table II; for the remaining instance (*i.e.,* 'SPIN-S'), *AutoCCAG* and *TCA* find the CCAs with the smallest and averaged sizes of both 1,174 (besides, *CASA* performs worse than *AutoCCAG* and *TCA* on this instance), but the run time required by *AutoCCAG* (860.2 sec) is much less than *TCA* (2,987.9 sec).

The experimental results in Tables I and II provide evidence that our *AutoCCAG* approach performs much better than all its competitors and dramatically pushes forward the state of the art in 4-way and 5-way CCAG solving.

D. RQ4: Evaluating AutoCCAG with a shorter cutoff time for 4-way and 5-way CCAG

In order to analyze the efficiency of *AutoCCAG*, we conduct more experiments to study the performance of *AutoCCAG* with a shorter cutoff time for solving 4-way and 5-way CCAG. In the experiments related to this RQ, *AutoCCAG* is evaluated to solve 4-way and 5-way CCAG instances with a half of the standard cutoff time.

The results of *AutoCCAG* (with a half of the standard cutoff time, *i.e.,* 500 seconds for 4-way CCAG and 5,000 seconds for 5-way CCAG) and *TCA-opt* (with the full cutoff time, *i.e.,* 1,000 seconds for 4-way CCAG and 10,000 seconds for 5-way CCAG) for solving 4-way and 5-way CCAG on the Real-world and IBM benchmarks are reported in Tables III and IV, respectively. From Table III, for solving 4-way CCAG, on both metrics of 'smallest size' and 'averaged size', *AutoCCAG* achieves better or equal performance compared to *TCA-opt* on all instances. Also, from Table IV, for solving 5-way CCAG, on both metrics of 'smallest size' and 'averaged size', *AutoCCAG* achieves better or equal performance

compared to *TCA-opt* on all instances but two. As shown in Table II, for those two instances ('Healthcare2' and 'NetworkMgmt'), in terms of 'smallest size' and 'averaged size', *AutoCCAG* with the full cutoff time achieves better or equal performance compared to *TCA-opt* with the full cutoff time.

The experimental results in Tables III and IV provide evidence that *AutoCCAG* with even a half of cutoff time can perform much better than *TCA-opt* with full cutoff time for solving 4-way and 5-way CCAG, which indicates that *AutoCCAG* requires much less run time to perform better than *TCA-opt* on solving the majority of CCAG instances.

E. Threats to Validity

There are three potential threats to validity of our evaluation:

Cutoff time: A potential threat to validity in our experiments is the cutoff time we set for each algorithm for solving 4-way CCAG. Following the existing work [14], we set the cutoff time to 1,000 seconds. According to the results reported in Table I, *AutoCCAG* can find CCAs for all instances except two ('Apache' and 'GCC'), which indicates that the cutoff time is reasonable. Nevertheless, the cutoff time might not be long enough for all experiments. To reduce this threat, we conduct additional experiments to run *AutoCCAG* and its competitors to solve 'Apache' and 'GCC', with the cutoff time of 10,000 seconds. The results in Table I show that *AutoCCAG* is still able to take much less computation time to find much smaller-sized CCAs compared to all competing CCAG algorithms on these 2 instances. For solving 5-way CCAG, the cutoff time for each algorithm run is set to 10,000 CPU seconds as the 5-way CCAG problem is more challenging and requires more computations. In our future work, we will design methods for recommending optimal cutoff time for the CCAG problem.

General t-way coverage: Although in this paper we only show the effectiveness of *AutoCCAG* through experiments on 4-way and 5-way CCAG, in fact, *AutoCCAG* is able to deal with general t-way CCAG as well. For example, *AutoCCAG* is able to deal with 2-way and 3-way CCAG. Due to limited space, we do not report our empirical results for 2-way and 3-way CCAG in this paper, but they are available online.[3] Actually, on the metrics of 'smallest size' and 'averaged size', the performance of *AutoCCAG* is better than or equal to that of all its state-of-the-art competitors (*i.e., TCA*, *CASA*, *HHSA* and *CHiP*) on all instances in the `Real-world` and `IBM` benchmarks for solving 2-way and 3-way CCAG. Furthermore, *AutoCCAG* supports other coverage criteria such as 6-way coverage. We will evaluate the effectiveness of *AutoCCAG* in t-way coverage ($t \geqslant 6$) in our future work.

Training set: According to Section IV-A, the training set used in our experiments is the `Synthetic` benchmark, which consists of 30 instances, and a potential threat to validity of our experiments is the small training set. As introduced in Section III, GP is the main machine learning model underlying *AutoCCAG*, and supports small training set [66]. Besides, a recent study [41] shows that using a small training set can achieve the state-of-the-art performance in solving the problem of minimum vertex cover, a well-known NP-hard combinatorial optimization problem. Furthermore, as described in Section IV-A, the `Synthetic` benchmark resembles the `Real-world` benchmark, and there is no explicit relationship between the `Synthetic` benchmark and the `IBM` benchmark. However, according to Tables I and II, *AutoCCAG* (trained on the `Synthetic` benchmark) performs best on all instances in the `IBM` benchmark (which covers extensive applications), indicating the generality of *AutoCCAG*.

VI. Related Work

Combinatorial interaction testing (CIT) is an important research topic in software engineering, and has been well explored for the last two decades. For the general information (*e.g.,* theoretical work and practical achievement), interested readers can refer to the book written by Kuhn *et al.* [67] and the survey summarized by Nie and Leung [20].

Practical algorithms for solving CCAG can be classified into three main categories: greedy algorithms, meta-heuristic algorithms and constraint-encoding algorithms. Greedy algorithms can rapidly generate a CCA in some scenarios where the metric of size is not the primary objective. Popular greedy algorithms can be categorized into two main classes: one-test-at-a-time (OTAT) algorithms and in-parameter-order (IPO) algorithms. The well-known algorithm *AETG* is the first one using the OTAT strategy [6]. Bryce *et al.* proposed a generic framework of the *AETG*-like algorithm [68]. Afterwards a number of variations of *AETG* were proposed [12], [16], [69]–[71]. The *IPO* algorithms extended horizontally and vertically to cover the tuples [7], [72]. Many variations of the *IPO* algorithms were also proposed (*e.g.,* [73], [74]).

Meta-heuristic algorithms work in an iterative manner: during the search process, those algorithms aim at seeking a CCA with a particular size k; once a k-sized CCA is found, then the algorithms will try to seek a CCA with the size smaller than k. Meta-heuristic algorithms include tabu search [14], [70], [75]–[78], simulated annealing [8], [15], [79]–[81], genetic algorithm [82], [83], *etc*. Actually, besides CCAG, meta-heuristic algorithms have exhibited success in solving various NP-hard problems [84]–[88].

Based on simulated annealing CCAG algorithms [60], [61], Garvin *et al.* proposed the one-sided narrowing and t-set replacement techniques [8], [10], resulting in an influential CCAG algorithm called *CASA* [8], which reduces runtime and finds smaller-sized CCAs. Jia *et al.* proposed a CCAG algorithm named *HHSA* [15], which uses hyper-heuristic search and dynamically applies different strategies during the search. Lin *et al.* proposed effective meta-heuristic CCAG algorithms dubbed *TCA* [14] and *FastCA* [77], which use tabu search to reduce the number of uncovered valid tuples, in order to improve the performance for solving CCAG. Recently, Mercan *et al.* presented an effective, parallel CCAG algorithm called *CHiP* [18], which can use vast amount of graphics processing units to implement the parallelism.

In addition, there is another way to tackle this problem. Banbara *et al.* [9] and Yamada *et al.* [13] encoded the CCAG

problem into the SAT problem and then use powerful constraint solvers to handle the resulting SAT-encoded instance. Yamada *et al.* proposed a constraint-encoded algorithm called *Calot* [13], which shows effectiveness for solving 2-way CCAG. More particularly, it can prove the optimality for 2-way CCAG on a number of instances [13]. However, solving t-way CCAG ($t \geqslant 3$) still remains a challenge for constraint-encoding algorithms.

VII. Conclusion

In this paper, we propose a novel, automated CCAG approach dubbed *AutoCCAG*, which is able to leverage the powerful automated configuration and automated selection techniques for solving the challenging CCAG problem. Extensive experiments on a broad range of real-world instances demonstrate that our *AutoCCAG* approach significantly outperforms its state-of-the-art CCAG competitors for solving 4-way and 5-way CCAG on public, real-world application benchmarks. Also, the performance of *AutoCCAG* is better than or equal to that of all its state-of-the-art CCAG competitors for solving 2-way and 3-way CCAG on these public benchmarks.

The testing benchmarks used in our experiments and the detailed experimental results (including the experimental results of all CCAG algorithms for solving 2-way, 3-way, 4-way and 5-way CCAG on all testing instances) are available at **https://github.com/chuanluocs/AutoCCAG**.

References

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

[2] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *Software and Systems Modeling*, vol. 18, no. 1, pp. 499–521, 2019.

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

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

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

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

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

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

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

[10] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Evaluating improvements to a meta-heuristic search for constrained interaction testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 61–102, 2011.

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

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

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

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

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

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

[17] P. Galinier, S. Kpodjedo, and G. Antoniol, "A penalty-based tabu search for constrained covering arrays," in *Proceedings of GECCO 2017*, 2017, pp. 1288–1294.

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

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

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

[21] C. Song, A. A. Porter, and J. S. Foster, "iTree: Efficiently discovering high-coverage configurations using interaction trees," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, 2014.

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

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

[24] R. Huang, H. Chen, Y. Zhou, T. Y. Chen, D. Towey, M. F. Lau, S. Ng, R. Merkel, and J. Chen, "Covering array constructors: An experimental analysis of their interaction coverage and fault detection," *The Computer Journal*, 2020.

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

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

[27] V. Okun, P. E. Black, and Y. Yesha, "Testing with model checker: Insuring fault visibility," *WSEAS Transactions on Systems*, vol. 2, no. 1, pp. 77–82, 2003.

[28] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, 2009.

[29] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *Proceedings of CP 2009*, 2009, pp. 142–157.

[30] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, "F-Race and iterated F-Race: An overview," in *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, 2010, pp. 311–336.

[31] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Proceedings of LION 2011*, 2011, pp. 507–523.

[32] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney, "Model-based genetic algorithms for algorithm configuration," in *Proceedings of IJCAI 2015*, 2015, pp. 733–739.

[33] M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari, "The irace package: Iterated racing for automatic algorithm configuration," *Operations Research Perspectives*, vol. 3, pp. 43–58, 2016.

[34] N. Dang, L. P. Cáceres, P. D. Causmaecker, and T. Stützle, "Configuring irace using surrogate configuration benchmarks," in *Proceedings of GECCO 2017*, 2017, pp. 243–250.

[35] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, 2008.

[36] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "ISAC – Instance-specific algorithm configuration," in *Proceedings of ECAI 2010*, 2010, pp. 751–756.

[37] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm selection and scheduling," in *Proceedings of CP 2011*, 2011, pp. 454–469.

[38] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm portfolios based on cost-sensitive hierarchical clustering," in *Proceedings of IJCAI 2013*, 2013, pp. 608–614.

[39] M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub, "AutoFolio: An automatically configured algorithm selector," *Journal of Artificial Intelligence Research*, vol. 53, pp. 745–778, 2015.

[40] M. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Schaub, "Automatic construction of parallel portfolios via algorithm configuration," *Artificial Intelligence*, vol. 244, pp. 272–290, 2017.

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

[42] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms," in *Proceedings of KDD 2013*, 2013, pp. 847–855.

[43] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Proceedings of NIPS 2015*, 2015, pp. 2962–2970.

[44] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Proceedings of IJCAI 2015*, 2015, pp. 3460–3468.

[45] C. Ansótegui, J. Pon, M. Sellmann, and K. Tierney, "Reactive dialectic search portfolios for MaxSAT," in *Proceedings of AAAI 2017*, 2017, pp. 765–772.

[46] H. Hoos, M. T. Lindauer, and T. Schaub, "claspfolio 2: Advances in algorithm selection for answer set programming," *Theory and Practice of Logic Programming*, vol. 14, no. 4-5, pp. 569–585, 2014.

[47] H. H. Hoos, "Programming by optimization," *Communications of the ACM*, vol. 55, no. 2, pp. 70–80, 2012.

[48] C. Luo, H. H. Hoos, and S. Cai, "PbO-CCSAT: Boosting local search for satisfiability using programming by optimisation," in *Proceedings of PPSN 2020*, 2020, pp. 373–389.

[49] F. Hutter, M. López-Ibáñez, C. Fawcett, M. T. Lindauer, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "AClib: A benchmark library for algorithm configuration," in *Proceedings of LION 2014*, 2014, pp. 36–40.

[50] J. Mockus, *Bayesian Approach to Global Optimization: Theory and Applications*. Kluwer Academic Publishers, 1989.

[51] D. J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[52] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[53] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of Bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

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

[55] F. Hutter, M. Lindauer, A. Balint, S. Bayless, H. H. Hoos, and K. Leyton-Brown, "The configurable SAT solver challenge (CSSC)," *Artificial Intelligence*, vol. 243, pp. 1–25, 2017.

[56] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proceedings of AAAI 2010*, 2010.

[57] A. Fréchette, N. Newman, and K. Leyton-Brown, "Solving the station repacking problem," in *Proceedings of AAAI 2016*, 2016, pp. 702–709.

[58] L. Xu, A. R. KhudaBukhsh, H. H. Hoos, and K. Leyton-Brown, "Quantifying the similarity of algorithm configurations," in *Proceedings of LION 2016*, 2016, pp. 203–217.

[59] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "Evaluating component solver contributions to portfolio-based algorithm selectors," in *Proceedings of SAT 2012*, 2012, pp. 228–241.

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

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

[62] I. Segall, R. Tzoref-Brill, and E. Farchi, "Using binary decision diagrams for combinatorial test design," in *Proceedings of ISSTA 2011*, 2011, pp. 254–264.

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

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

[65] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proceedings of NIPS 2012*, 2012, pp. 2960–2968.

[66] J. C. Platt, C. J. C. Burges, S. Swenson, C. Weare, and A. Zheng, "Learning a Gaussian process prior for automatically generating music playlists," in *Proceedings of NIPS 2001*, 2001, pp. 1425–1432.

[67] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC press, 2013.

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

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

[70] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Software Testing, Verification and Reliability*, vol. 17, no. 3, pp. 159–182, 2007.

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

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

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

[74] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[75] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete Applied Mathematics*, vol. 138, no. 1-2, pp. 143–152, 2004.

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

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

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

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

[80] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, "Variable strength interaction testing of components," in *Proceedings of COMPSAC 2003*, 2003, p. 413.

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

[82] S. A. Ghazi and M. A. Ahmed, "Pair-wise test coverage using genetic algorithms," in *Proceedings of CEC 2003*, vol. 2, 2003, pp. 1420–1424.

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

[84] C. Luo, S. Cai, W. Wu, and K. Su, "Double configuration checking in stochastic local search for satisfiability," in *Proceedings of AAAI 2014*, 2014, pp. 2703–2709.

[85] C. Luo, K. Su, and S. Cai, "More efficient two-mode stochastic local search for random 3-satisfiability," *Applied Intelligence*, vol. 41, no. 3, pp. 665–680, 2014.

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

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

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