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 \( t \geq 4 \) 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 \((t \geq 4)\) still remains a challenge for meta-heuristic algorithms [18], [21].

It is important to solve \( t \)-way CCAG with \( t \geq 4 \). Much work (e.g., [4], [19], [22]–[24]) has provided evidence that higher
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 \((t \geq 4)\). Another recent work [24] demonstrates that, through extensive empirical study on various software, the fault detection rate of \(t\)-way CCAG with \(t \geq 4\) 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 \(t \geq 4\) 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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{AutoCCAG}, which leverages the effectiveness of automated configuration and automated selection techniques.
- We perform extensive experiments, demonstrating that \textit{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.

\textit{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 \textit{tuple} and \textit{test case}, as described below.

\textit{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.

\textit{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 \(\tau\) is covered by test case \(tc\) if, and only if, \(\tau \subseteq tc\), that is, the options in \(\tau\) 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.

\textit{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 \textit{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.

1) Workflow of AutoCCAG: As illustrated in Figure 1, AutoCCAG works as follows: Given a target, configurable CCAG algorithm, 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 $\leq t \leq 5$). 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,
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: \( IB\_PCG \): iteration budget for PCG; \( TB\_BOAC \): time budget for BOAC; \( TI \): a set of training instances; \( a \): the target algorithm;

Output: \( PC \): a set of promising configurations of \( a \);

1. \( PC \leftarrow \emptyset \);
2. while time budget \( IB\_PCG \) is not reached and \( TI \neq \emptyset \) 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:

\[
amc(b) = \ln \frac{size(PC \setminus \{b\})}{size(PC)} \tag{1}
\]

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.
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 $listmc$. After the final scheduling plan $listmc$ is obtained, our AutoCCAG approach runs the target CCAG algorithm with the final scheduling plan $listmc$ 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. 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].

IBM. This real application benchmark is comprised of 20 practical instances, originally introduced in the literature [62], and is available online. These instances are generated aiming to provide 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].

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 (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?

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

5https://github.com/chuanluocs/AutoCCAG

In this RQ, 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].


CASA [8] is a high-performance simulated annealing CCAG algorithm. 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.

We note that HHSA [15] is an effective CCAG algorithm for 2-way and 3-way CCAG. Since the implementation of HHSA available online does not support solving t-way CCAG with t ≥ 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, 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, 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.

We note that HHSA [18] is an effective CCAG algorithm for 2-way and 3-way CCAG. Since the implementation of HHSA available online does not support solving t-way CCAG with t ≥ 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, and AutoCCAG can find smaller-sized or equal-sized CCAs than HHSA on all instances for 2-way and 3-way CCAG.
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
configuration. Hence, \textit{TCA-opt} does not leverage the effectiveness of automated selection. The comparative results of \textit{AutoCCAG} and \textit{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 \textit{TCA-opt}, our \textit{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.

\textbf{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 \textit{AutoCCAG} approach against existing state-of-the-art CCAG algorithms, in order to show the effectiveness of \textit{AutoCCAG}.

The experimental results of \textit{AutoCCAG} and its state-of-the-art competitors (i.e., \textit{TCA}, \textit{CASA} and \textit{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, \textit{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’, \textit{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’), \textit{AutoCCAG}, \textit{TCA} and \textit{CHiP} can find the CCA with the same smallest size of 5,610 (besides, \textit{CASA} performs worse than \textit{AutoCCAG}, \textit{TCA} and \textit{CHiP} on this instance), but the run time required by \textit{AutoCCAG} (233.8 sec) is much less than \textit{TCA} (561.1 sec) and \textit{CHiP} (207.136 sec). Also, on the metric of ‘averaged size’, \textit{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’, \textit{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’), \textit{AutoCCAG} and \textit{TCA} find the CCAs with the smallest and averaged sizes of both 1,174 (besides, \textit{CASA} performs worse than \textit{AutoCCAG} and \textit{TCA} on this instance), but the run time required by \textit{AutoCCAG} (860.2 sec) is much less than \textit{TCA} (2,987.9 sec).

The experimental results in Tables I and II provide evidence that our \textit{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.

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

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

The results of \textit{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 \textit{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’, \textit{AutoCCAG} achieves better or equal performance compared to \textit{TCA-opt} on all instances. Also, from Table IV, for solving 5-way CCAG, on both metrics of ‘smallest size’ and ‘averaged size’, \textit{AutoCCAG} achieves better or equal performance.

\begin{table}[h]
\centering
\caption{Comparing AutoCCAG (with the cutoff time of 5,000 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 seconds.}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Instance} & \textbf{AutoCCAG (5,000 sec)} & \textbf{TCA-opt (1,000 sec)} & \\
\hline
& \textbf{min (avg) time} & \textbf{min (avg) time} & \\
\hline
Apache & 136 (136.9) & 168 (138.6) & 637.6 \\
Bugzilla & 678 (680.7) & 168 (168.6) & 637.6 \\
GCC & 308 (308.0) & 308 (308.0) & 121.6 \\
SPIN-S & 1119 (1124.1) & 1526 (1526.2) & 937.6 \\
SPIN-V & 161 (168.8) & 343.3 & 376.6 \\
Healthcare2 & 728 (731.8) & 397.2 & 851.6 \\
Healthcare3 & 1351 (1335.6) & 476.9 & 987.1 \\
Healthcare4 & 75361 (75361.0) & 165.4 & 992.1 \\
NetworkMgmt & 35610 (35610.0) & 233.8 & 337.7 \\
ProcessorComm1 & 485 (488.7) & 381.5 & 461.2 \\
ProcessorComm2 & 375 (376.5) & 298.7 & 517.3 \\
Services & 6406 (6408.8) & 484.3 & 853.4 \\
Storage4 & 5518 (5523.5) & 481.3 & 993.8 \\
Storage5 & 11041 (11053.3) & 452.8 & 971.0 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\caption{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 seconds.}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Instance} & \textbf{AutoCCAG (5,000 sec)} & \textbf{TCA-opt (10,000 sec)} & \\
\hline
& \textbf{min (avg) time} & \textbf{min (avg) time} & \\
\hline
Apache & 566 (576.8) & 168 (168.6) & 637.6 \\
Bugzilla & 1174 (1174.0) & 860.2 & 2908.4 \\
SPIN-S & 6429 (6809.1) & 3231.8 & 3376.0 \\
SPIN-V & 456781 (457835.1) & 4999.8 & 9900.7 \\
Healthcare2 & 521 (522.9) & 3605.9 & 6422.9 \\
Healthcare3 & 3210 (3221.5) & 4476.7 & 9527.0 \\
Healthcare4 & 6973 (6994.0) & 4928.8 & 9944.8 \\
Insurance & 456781 (457835.1) & 4999.8 & 9900.7 \\
NetworkMgmt & 24773 (24788.6) & 4963.9 & 9874.4 \\
ProcessorComm1 & 2038 (2040.0) & 3560.3 & 7643.3 \\
ProcessorComm2 & 2506 (2511.3) & 3582.7 & 9600.1 \\
Services & 32987 (33243.9) & 4694.4 & 8980.9 \\
Storage4 & 34207 (34261.6) & 4966.9 & 9939.4 \\
Storage5 & 66857 (66966.8) & 4992.9 & 9457.1 \\
\hline
\end{tabular}
\end{table}
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. 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 ($t \geq 6$) 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 ($t \geq 3$) 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


