A Benchmark Generator for Combinatorial Testing

Carlos Ansótegui[1][0000−0001−7727−2766] and Eduard Torres[1][0000−0002−3136−7513]

[1] Logic & Optimization Group, University of Lleida
[2] Lleida, Spain. {carlos.ansotegui,eduard.torres}@udl.cat

Abstract. Combinatorial Testing (CT) tools are widely used to test complex systems such as train control, graphical user interfaces, and autonomous driving software. Despite active research on CT techniques, there is a notable lack of benchmarks for systematically evaluating CT tools in terms of correctness, effectiveness, and efficiency. This paper introduces a novel CT benchmark generator that leverages the structure of established combinatorial problems from other research communities to produce realistic and diverse benchmarks. We also conduct an extensive empirical evaluation of CT tools using the generated benchmarks, providing insights into the conditions under which different CT tools perform best.

Keywords: Combinatorial Testing · Benchmark Generator · Covering Arrays · Satisfiability.

1 Introduction

Technology is essential to modern life, yet software systems inevitably contain bugs and faults. As a result, system designers aim to detect and fix as many faults as possible through effective testing before deployment. Combinatorial Testing (CT) [13] addresses this challenge by efficiently overlapping combinations of input parameters of a System Under Test (SUT). CT is based on the observation that most faults are caused by interactions among a relatively small number of parameters [12], allowing to detect many bugs with a manageable number of tests.

A major challenge in developing CT tools is the lack of comprehensive benchmark suites for evaluating their correctness and performance. This issue is particularly critical for CT tools that handle constraints among SUT parameters, as real-world constrained benchmarks are scarce, and artificially generated ones often lack realism or controlled hardness. Existing benchmarks are limited: five real-world and thirty synthetic benchmarks in [10], twenty relatively easy real-world benchmarks in [15], a few additional cases in [20, 18], and 116 variants of 48 real-world benchmarks in [17]. This scarcity interferes with fair tool comparison, slows algorithmic progress, introduces benchmark bias, and complicates the organization of competitions, which have historically driven advances in other

areas[3]. These issues are especially relevant when compared to other related fields such as the Satisfiability research community [8], where new real-world benchmarks are consistently submitted to the anual competitions.

In this paper, we introduce a novel generator for constrained SUT benchmarks based on SAT instances. Our approach derives SUTs as *subproblems* of existing SAT formulations by fixing variables and simplifying the formula using incomplete inference techniques such as unit propagation. From the resulting subformula, a subset of variables is selected as SUT input parameters, while the remaining variables are treated as auxiliary. This method naturally leverages the extensive collection of industrial, crafted, and random SAT benchmarks, enabling access to diverse and realistic constraint structures. Furthermore, we demonstrate how the hardness of the generated constraints can be empirically controlled, addressing a limitation of existing benchmarks, which are often *easy* in practice. In this initial version, we focus on SUTs with Boolean parameters to study constraint complexity, with plans to extend the generator to mixed-domain parameters in future work, as discussed in Section 5.

This paper is structured as follows. Section 2 presents the basic CT and SAT definitions and concepts that we use in this work. Section 3 introduces *SUT-G*, our approach for generating constrained SUT benchmarks. In Section 4, we evaluate benchmarks produced by *SUT-G* using two state-of-the-art algorithms for Mixed Covering Arrays with Constraints (MCAC), offering insights into their behavior. Finally, Section 5 concludes the paper.

2 Preliminaries

This section introduces basic concepts about Combinatorial Testing (CT) and Satisfiability (SAT). We start introducing the concepts related to CT.

A System Under Test (SUT) model is a tuple $\langle P, \varphi \rangle$, where P is a finite set of variables p of finite domain, called SUT parameters, and φ is a set of constraints on P, called SUT constraints, that implicitly represents the parameterizations that the system accepts. We denote by $d(p)$ and g_p, respectively, the domain and the cardinality domain of p. For the sake of clarity, we will assume that the system accepts at least one parameterization. In the following, we assume $S = \langle P, \varphi \rangle$ to be a SUT model. We will refer to P as S_P, and to φ as S_φ.

An assignment is a set of pairs (p, v) where p is a variable and v is a value of the domain of p. A test case for S is a full assignment A to the variables in S_P such that A entails S_φ (i.e. $A \models S_\varphi$). A parameter tuple of S is a subset $\pi \subseteq S_P$. A value tuple of S is a partial assignment to S_P; in particular, we refer to a value tuple of length t as a t-tuple.

A t-tuple τ is forbidden if τ does not entail S_φ (i.e. $\tau \models \neg S_\varphi$). Otherwise, it is allowed. We refer to the set of allowed t-tuples as $\mathcal{T}_a = \{\tau \mid \tau \not\models \neg S_\varphi\}$.

A test case υ *covers* a value tuple τ if both assign the same domain value to the variables in the value tuple, i.e., $\upsilon \models \tau$. A test suite Υ *covers* a value tuple

[3] The CT community has recently organized also a competition: https://fmselab.github.io/ct-competition/Competition2022.html.

τ (i.e., $\tau \subseteq \Upsilon$) if there exist a test case $\upsilon \in \Upsilon$ s.t. $\upsilon \models \tau$. We refer to $\upsilon \not\models \tau$ ($\tau \not\subseteq \Upsilon$) when a test case (test suite) *does not cover* τ.

A Mixed Covering Array with Constraints (MCAC), denoted by $CA(N; t, S)$, is a set of N test cases for a SUT model S such that all t-tuples are at least covered by one test case. The term *Mixed* reflects that the domains of the parameters in S_P are allowed to have different cardinalities. The term *Constraints* reflects that S_φ is not empty. The MCAC problem is to find an MCAC of size N. The Covering Array Number, $CAN(t, S)$, is the minimum N for which there exists an MCAC $CA(N; t, S)$. The Covering Array Number problem is to find an MCAC of size $CAN(t, S)$.

Next, we present several definitions regarding the SATisfiability (SAT) problem [8]. A literal is a propositional variable x or a negated propositional variable $\neg x$. A clause is a disjunction of literals. A Conjunctive Normal Form (CNF) formula, also called SAT instance, is a conjunction of clauses.

A truth assignment for an instance φ is a mapping that assigns to each propositional variable in φ either 0 (False) or 1 (True). A truth assignment is *partial* if the mapping is not defined for all the propositional variables in φ.

A truth assignment I satisfies a literal x ($\neg x$) if I maps x to 1 (0); otherwise, it is falsified. A truth assignment I satisfies a clause if I satisfies at least one of its literals; otherwise, it is violated or falsified. Given a partial truth assignment I, a literal or a clause is undefined if it is neither satisfied nor falsified. A clause c is a unit clause under I if c is not satisfied by I and contains exactly one undefined literal.

A SAT instance φ is satisfiable if there is a truth assignment I, called model, such that I satisfies all the clauses in φ. Otherwise, φ is unsatisfiable. The Satisfiability (SAT) problem SAT problem is to determine whether a SAT instance is satisfiable.

We refer to SAT solver as the implementation of an algorithm that takes as input a SAT instance and decides the SAT problem. The solver is said to be incremental if the input SAT instance can be modified and solved again while reusing some information from previous steps.

Given a SAT instance φ and a partial truth assignment I, we refer as Unit Propagation, denoted by $UP(I, \varphi)$, to the Boolean inference mechanism (propagator) defined as follows: Find a unit clause in φ under I, where l is the undefined literal. Then, propagate the unit clause, i.e. extend I with $x = 1$ ($x = 0$) if $l \equiv x$ ($l \equiv \neg x$) and repeat the process until a fixpoint is reached or a conflict is derived (i.e. a clause in φ is falsified by I). We refer to $UP(I, \varphi)$ simply as $UP(\varphi)$ when I is empty.

Algorithm SATSolver presents the interface of a modern incremental SAT solver. Clauses are added using add_clause. The *solve* method determines satisfiability, returning SAT or UNSAT, and sets *model* or *core* accordingly, along with the number of conflicts $n\_conflicts$ generated during the call. The method can also take a set of assumptions $assumps$, which temporarily fix the truth values of the corresponding literals during solving.

The *propagate* method performs unit propagation on the current formula, optionally under a set of assumptions, and returns the propagated literals. Finally, *set_seed* initializes the solver's random number generator with the specified *seed*.

Code SATSolver: Interface of a modern incremental SAT solver

```
   #Attributes
1  n_vars #number of variables of the formula loaded
2  n_conflicts #number of conflicts of the last call to solve
3  core #last core found
4  model #last model found
   #Methods
5  function add_clause(c : clause)
       #Adds the clause c to the solver
6  function solve(assumps : literals)
       #If formula is satisfiable, status ← SAT, sat.model is updated
       #If formula is unsatisfiable, status ← UNSAT, sat.core is updated
       #If assumps ≠ ∅, sets each literal in assumps to the solver trail
7      return status
8  function propagate(assumps : literals)
       #Performs Unit Propagation over the input formula given assumps
9      return The set of propagated literals
10 function set_seed(seed : int)
       #Sets the Random Number Generator seed of the SAT solver to the given seed
```

3 Generating SUTs with Constraints

In this section, we present the *SUT-G* benchmark generator, which generates Systems Under Test (SUT) models with constraints from any SAT instance.

SUT-G generates SUT models with constraints that include auxiliary variables. These variables are commonly introduced to avoid combinatorial explosion when translating constraints into Conjunctive Normal Form (CNF). For example, the Tseitin transformation introduces fresh auxiliary variables when converting non-CNF circuits into CNF [16]. Auxiliary variables are also widely used in CNF encodings of cardinality and pseudo-Boolean constraints [11]. Their use enables translations that are both tractable in size and arc-consistent, and has been shown to improve SAT solver performance [3].

Since existing SUT model formats do not support auxiliary variables, we propose a modified format extending *ACTS* to explicitly represent such constraints. *SUT-G* outputs generated SUTs in this format, which we call *Extended ACTS*.

Algorithm SUT-G presents the pseudocode of the proposed *SUT-G* generator. The generator takes a SAT formula φ as input and produces a SUT model

Algorithm SUT-G: SUT Generation algorithm from SAT instances

Input: SAT formula φ, Number of SUT input parameters n
Parameters: Max conflicts c_{max}, Min conflicts c_{min}, Assumptions increase $\Delta_{\mathcal{A}}$, Assumptions decrease $\nabla_{\mathcal{A}}$, Seeds $\mathcal{S}$, Maximum number of tries MAX_TRIES
Output: SUT model S

1 $\langle$status, $\mathcal{A}\rangle \leftarrow find\_satisfiable\_subproblem($ φ, n, c_{max}, c_{min}, $\Delta_{\mathcal{A}}$, $\nabla_{\mathcal{A}}$, $\mathcal{S}$, MAX_TRIES)
2 **if** *status* = *FAIL* **then return** *None*
`# Generate the constraints of the SUT`
3 **for** $lit \in \mathcal{A}$ **do**
`# Add each literal in` $\mathcal{A}$ `to` φ `as unit clause`
4 $\varphi \leftarrow \varphi \cup \{\{lit\}\}$
5 $\varphi' \leftarrow simplify(\varphi)$
`# Select the input parameters of the SUT`
6 $P \leftarrow sample(\varphi'.vars, n)$
7 $SUT \leftarrow \langle P, \varphi' \rangle$
8 **return** SUT

with constraints, where φ is adapted to a target difficulty Difficulty is measured by the average number of conflicts required by a SAT solver to solve the instance. In addition to φ and the desired number of SUT model parameters n, *SUT-G* receives bounds on the acceptable number of conflicts (c_{min} and c_{max}), as well as additional control parameters that will be explained later ($\Delta_{\mathcal{A}}$, $\nabla_{\mathcal{A}}$, $\mathcal{S}$, and MAX_TRIES).

At a high level, *SUT-G* searches for a satisfiable subproblem of φ that meets the specified constraints (line 1). If the formula is unsatisfiable or no subproblem satisfies the requirements, the algorithm terminates with failure. Otherwise, it returns a set of assumptions that adjust φ to match the target hardness.

These assumptions are applied to φ as unit clauses (lines 3-4), after which the formula is simplified to obtain φ' (line 5). Simplification propagates unit clauses, removes satisfied clauses, deletes falsified literals, and renames variables to ensure a compact index range from 1 to $\varphi'.n_vars$.

To finalize the SUT model generation, the algorithm randomly selects n variables from φ' as SUT parameters (line 6). The remaining variables are treated as auxiliary.

Currently, *SUT-G* generates SUTs with boolean parameters. Section 5, outlines extensions to midex-domain SUTs.

Algorithm SUT-G (Auxiliary functions) shows the auxiliary functions used by the *SUT-G* generator.

Function `find_satisfiable_subproblem` (called in line 1 of Algorithm SUT-G) initializes an incremental SAT solver with the input formula φ and an empty set of assumptions $\mathcal{A}$ (lines 2–3). It then calls `solve_subproblem` to compute the subproblem's *hardness*, measured as the number of conflicts c, for φ under

Algorithm SUT-G (Auxiliary functions): Auxiliary functions for SUT-G

```
1  function find_satisfiable_subproblem(φ, n, c_max, c_min, Δ_A, ∇_A, S, MAX_TRIES)
2      sat ← incremental SAT solver initialized with φ
3      A ← ∅
4      ⟨model, c⟩ ← solve_subproblem(sat, A, S)
5      if model = ∅ or sat.n_vars − |sat.propagate(A)| < n or c < c_min then
           return ⟨ FAIL, _ ⟩
6      tries ← 1
7      while tries < MAX_TRIES do
8          if sat.n_vars − |sat.propagate(A)| < n or c < c_min then
9              if |A| = 0 then return ⟨ FAIL, _ ⟩
10             A ← A \ sample(A, ∇_A)
11         elif c > c_max then A ← A ∪ sample(model \ A, Δ_A)
12         else return ⟨ SUCCESS, sat.propagate(A) ⟩
13         if sat.n_vars − |sat.propagate(A)| ≥ n then
14             ⟨model, c⟩ ← solve_subproblem(sat, A, S)
15             tries ← tries + 1
16     return ⟨ FAIL, _ ⟩
17
18 function solve_subproblem(sat, A, S)
19     c ← 0
20     models ← ∅
21     for seed ∈ S do
22         sat.set_seed(seed)
23         if sat.solve(A) = UNSAT then return ⟨ _, ∅, _ ⟩
24         models ← models ∪ {sat.model}
25         c ← c + sat.n_conflicts
26     return ⟨sample(models, 1), c/|S|⟩
```

$\mathcal{A}$ (line 4). This call also returns a model used in subsequent iterations. Since $\mathcal{A}$ is initially empty, this first call corresponds to solving φ itself.

The function immediately returns *fail* if any of the following holds: (i) φ is UNSAT (indicated by an empty model), (ii) there are fewer than n *unfixed* variables available to serve as SUT parameters (counting variables fixed by unit propagation under $\mathcal{A}$ as fixed) or (iii) the problem is too easy ($c < c_{min}$) (line 5).

The main loop (lines 6 - 15) attempts up to MAX_TRIES iterations to find a suitable subproblem. Exceeding this limit results in *fail* (line 16). Each iteration adjusts the assumption set $\mathcal{A}$ to obtain a subproblem with sufficient unfixed variables and hardness within the target conflict range.

Increasing $\mathcal{A}$ generally simplifies the subproblem, while decreasing it tends to make the problem harder. Although this greedy strategy is not guaranteed (since

additional constraints may sometimes increase hardness), we found it effective in practice (see Section 4.3).

If the current subproblem is too easy ($c < c_{min}$) or has too few unfixed variables, the algorithm attempts to decrease $\mathcal{A}$ by removing $\nabla_{\mathcal{A}}$ randomly chosen literals (line 10). If $\mathcal{A}$ is already empty, no further simplification is possible and the function returns *fail* (lines 9 - 9). This situation can arise due to clause learning in the incremental solver, which may permanently reduce the hardness of the original formula.

Conversely, if the subproblem is too hard ($c > c_{max}$), the algorithm increases $\mathcal{A}$ by adding $\Delta_{\mathcal{A}}$ random literals from the current model (line 11). If neither condition applies, the subproblem satisfies both the hardness and variable requirements, and the function returns *success* along with the propagated assumptions (line 12).

After each modification of $\mathcal{A}$, the algorithm recomputes the conflicts and model by calling `solve_subproblem` and increments the trial counter (lines 14 - 15). These steps are skipped when there are insufficient unfixed variables, in which case the algorithm continues decreasing $\mathcal{A}$ without increasing the number of tries.

Line 18 defines the function `solve_subproblem`, which solves a given subproblem and estimates its hardness in terms of SAT solver conflicts. The function initializes the conflict counter c and the set of models *models* to 0 and $\emptyset$, respectively (lines 19 - 20). It then iterates over the set of user-provided seeds $\mathcal{S}$ (line 21). For each seed, the SAT solver's random number generator is initialized accordingly (line 22), and the subproblem, which is defined by the current formula and the assumption set $\mathcal{A}$, is solved (line 23).

If the solver reports UNSAT for any seed, the function immediately returns an empty model (line 23). Otherwise, when the subproblem is SAT, the corresponding model is stored in *models* (line 24), and the number of conflicts is accumulated into c (line 25).

Finally, the function returns a randomly selected model from *models* together with the average number of conflicts over all seeds (line 26), which helps mitigate variability due to particularly lucky or unlucky solver runs.

4 Assessing MCAC Tools with *SUT-Gen*

In this section, we assess the quality of the SUT constraints generated by SUT-G. We empirically show that the SAT-based SUTs we generate are different from the random SUTs available in the CT competition (even using the same number of parameters) both in terms of runtime and size of the covering array. This experimental evaluation has three main research questions:

1. **RQ1**: Compare the SUT-G generator with other state-of-the-art random SUT generators.
2. **RQ2**: Analyze the impact of the generated SUT constraints with respect to the same unconstrained SUT.

3. **RQ3**: Evaluate and compare the differences in test suite size and runtime of the mentioned MCAC algorithms using the generated CT benchmarks, and compare their behaviour with the available CT benchmarks.

For benchmark generation, we used industrial SAT instances, aiming to preserve the structural properties of real-world constraints. By doing so, the SUT constraints generated by SUT-G retain industrial-like characteristics and provide a realistic approximation of constraints found in practical SUTs. We selected 165 instances from the SAT Competition 2009 [14], as these are generally easier for modern SAT solvers to handle[4]. This choice was made for convenience, since SUT-G can adjust instance hardness as described in Section 3. From this set, we retained only satisfiable instances that met the conflict thresholds described below.

Benchmarks were generated with 10, 50, and 100 binary parameters, using conflict limits $c_{max} \in \{5000, 10000\}$ and $c_{min} = c_{max}/2$ (see Section 3). In total, we generated 82 benchmarks, which will be released alongside the SUT-G generator once the paper is accepted. Tables 8, 9, and 10 (Appendix) summarize their characteristics, including the number of variables, clauses, and literals, as well as maximum and average clause sizes.

All our experiments were conducted on a cluster of nodes equipped with two AMD 7403 processors (24 cores at 2.8GHz) and 21GB of RAM per core. Results are reported in Tables 1, 2, 3, 4, 5, and 6. Best results in terms of average test suite size and runtime are highlighted in bold. Each benchmark was executed with five different random seeds to mitigate stochastic effects, using a time limit of 8h and a memory limit of 21GB. All benchmarks were evaluated for strengths $t = 2, 3, 4, 5$.

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-01	45.03	**8.70**	48.85	10.80	**46.98**	18.40	48.33	**18.10**	**48.29**	**44.80**	47.63	47.60	**46.59**	92.10	49.65	**88.90**	**49.07**
AProVE09-05	17.70	**6.50**	23.80	7.20	**21.96**	**11.60**	24.09	13.20	**22.92**	**14.30**	21.15	18.60	**20.06**	**15.00**	21.86	20.60	**21.01**
AProVE09-07	9.86	**4.00**	19.15	5.10	**17.21**	**4.00**	17.67	6.40	**17.41**	**4.00**	14.53	6.80	**13.70**	**4.00**	**14.58**	7.10	14.74
AProVE09-08	10.15	**1.00**	17.32	2.00	**15.68**	**1.00**	12.61	2.00	**12.45**	**1.00**	**12.68**	2.00	13.15	**1.00**	13.23	2.00	**13.16**
AProVE09-21	32.84	**8.80**	60.23	9.80	**55.44**	**22.40**	**81.05**	27.10	89.10	**49.40**	98.29	56.00	**89.96**	**94.80**	127.30	105.50	**103.04**
gss-13	32.48	**1.00**	38.84	2.00	**36.96**	**1.00**	40.25	2.00	**39.58**	**1.00**	37.95	2.00	**37.28**	**1.00**	40.77	2.00	**40.12**
gss-14	31.40	**1.00**	34.55	2.00	**33.26**	**1.00**	35.02	2.00	**34.70**	**1.00**	**31.10**	**1.00**	31.36	**1.00**	**31.58**	1.90	31.72
q_query_3_L60	70.43	**9.00**	**80.89**	10.30	81.21	**20.50**	**86.42**	25.90	86.60	**44.70**	88.06	52.80	**86.80**	**74.40**	94.60	89.60	**90.42**
q_query_3_l37	35.81	**8.70**	47.01	9.80	**42.25**	19.50	49.37	**18.80**	**45.02**	**44.90**	51.51	45.40	**44.31**	96.40	57.80	**93.20**	**48.64**
q_query_3_l38	34.50	**8.80**	43.10	9.20	**39.57**	21.40	46.63	**21.20**	**41.87**	49.60	45.99	**48.80**	**41.51**	101.00	50.22	**100.20**	**44.42**
q_query_3_l39	30.29	**7.90**	39.23	9.40	**38.54**	**17.60**	40.99	19.00	**40.09**	**37.30**	38.65	37.50	**38.65**	69.70	40.55	**69.30**	**39.86**
q_query_3_l40	38.53	**8.20**	46.13	8.80	**43.90**	**19.80**	47.06	20.20	**45.41**	43.80	45.39	**42.70**	**44.50**	86.10	48.04	**82.70**	**45.93**
q_query_3_l41	39.60	**7.50**	48.65	7.90	**46.81**	**16.30**	48.80	16.50	**48.09**	33.90	47.20	**32.50**	**46.39**	58.40	49.47	**56.30**	**47.90**
q_query_3_l42	38.85	**8.20**	45.25	8.40	**43.10**	17.40	46.51	**17.20**	**44.36**	40.90	43.91	**39.60**	**42.21**	82.10	45.57	**77.30**	**44.99**
q_query_3_l43	38.88	**6.60**	44.98	6.80	**43.30**	**11.90**	41.49	12.20	**40.74**	22.90	41.24	**22.80**	**40.42**	**32.00**	**41.06**	**32.00**	41.09
rpoc_xits_17	66.08	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
unconstr	1.60	**8.70**	**1.45**	10.80	1.54	18.40	**1.39**	**18.10**	1.41	**44.80**	**1.81**	47.60	1.85	92.10	**1.64**	**88.90**	1.96

Table 1. Average test suite size and runtime for the instances with 10 parameters and 5000 conflicts

[4] Modern SAT solvers typically solve instances from earlier competitions more efficiently.

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-05	17.01	**2.00**	18.94	3.00	**18.38**	**2.00**	19.01	3.90	**18.55**	**2.00**	19.08	3.90	**18.35**	**2.00**	19.02	4.00	**18.91**
AProVE09-07	11.12	**2.00**	18.60	2.80	**17.37**	**2.00**	16.27	3.60	**15.75**	**2.00**	16.72	3.80	**15.54**	**2.00**	16.94	3.70	**16.31**
AProVE09-15	96.45	**7.40**	**123.34**	8.20	125.76	**14.60**	132.44	16.90	**126.06**	**25.00**	136.57	27.00	**134.26**	**26.00**	151.29	33.00	**143.59**
AProVE09-20	35.41	**9.40**	**79.14**	12.20	109.54	**24.00**	**92.03**	31.30	108.23	**48.10**	**101.69**	57.60	105.22	**80.50**	**124.27**	95.10	124.57
UR-10	72.50	**4.00**	78.05	4.80	**77.91**	**4.00**	**78.48**	5.40	79.59	**4.00**	80.35	5.60	**77.73**	**4.00**	79.01	5.50	**77.55**
q_query_3_L60	74.17	**8.20**	116.93	9.40	**105.96**	**20.60**	139.06	23.40	**133.53**	**45.50**	154.81	55.20	**143.06**	**95.90**	176.27	111.90	**162.90**
q_query_3_l38	33.14	**8.20**	45.21	9.00	**44.14**	**19.60**	44.29	21.30	**43.86**	**44.00**	46.06	44.60	**43.96**	85.50	48.11	**84.20**	**46.87**
q_query_3_l39	36.82	**8.60**	48.03	8.90	**45.49**	19.70	48.59	**19.00**	**45.17**	42.40	51.56	**41.00**	**47.87**	85.10	54.43	**82.50**	**50.89**
q_query_3_l40	37.37	**8.20**	63.44	9.20	**58.56**	**19.80**	70.76	22.20	**62.97**	**43.80**	73.32	45.10	**67.89**	89.00	80.83	**88.10**	**78.54**
q_query_3_l41	37.01	**8.00**	**62.62**	9.00	63.77	**18.50**	63.25	18.70	**61.52**	40.60	**66.68**	**40.00**	71.32	83.00	**71.06**	**80.50**	73.97
q_query_3_l42	41.49	**8.20**	73.69	8.80	**64.57**	**19.00**	72.97	19.50	**72.16**	44.50	**74.83**	**44.40**	82.75	90.70	**77.65**	**88.50**	86.14
q_query_3_l43	41.09	**8.30**	144.50	9.40	**128.94**	**20.50**	**146.43**	21.40	169.98	**44.80**	**153.72**	45.60	179.46	**84.30**	164.93	**84.30**	**162.27**
rpoc_xits_17	78.84	**8.00**	**141.70**	-	-	-	-	-	-	-	-	-	-	-	-	-	-
unconstr	1.60	**8.70**	**1.45**	10.80	1.54	18.40	**1.39**	**18.10**	1.41	**44.80**	**1.81**	47.60	1.85	92.10	**1.64**	**88.90**	1.96

Table 2. Average test suite size and runtime for the instances with 10 parameters and 10000 conflicts

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-01	43.99	**13.60**	51.56	22.60	**50.03**	38.20	63.19	**37.70**	**53.12**	**98.30**	**103.07**	100.40	189.62	**258.60**	**293.92**	273.50	813.39
AProVE09-05	18.53	**15.40**	**24.87**	16.80	25.98	**33.70**	**31.97**	38.50	32.15	**55.30**	170.29	61.30	**154.08**	**67.70**	4176.49	82.90	**1281.04**
AProVE09-07	9.91	**12.10**	**23.97**	14.10	25.02	**20.00**	28.21	25.30	**27.48**	**24.20**	152.60	31.80	**113.77**	**25.00**	2843.57	36.50	**1311.89**
AProVE09-20	35.71	**30.80**	**157.24**	35.20	178.90	**155.00**	403.66	162.00	**277.87**	674.10	4271.89	**635.90**	**971.49**	-	-	**2495.40**	**6863.43**
AProVE09-21	33.15	**34.30**	**251.27**	44.00	328.55	**166.20**	730.20	195.10	**585.66**	**655.90**	5098.96	682.70	**1270.63**	-	-	**2218.50**	**7058.99**
q_query_3_L60	71.70	**35.00**	**110.07**	38.90	128.66	**138.00**	273.13	153.60	**188.72**	**419.50**	2590.08	442.50	**745.88**	**995.86**	18367.13	1041.50	**3484.97**
q_query_3_l37	34.35	**19.50**	**82.46**	25.90	95.55	**82.10**	**195.46**	113.70	240.51	**346.40**	795.43	412.00	**727.71**	1463.10	13022.73	**1370.90**	**2870.22**
q_query_3_l38	37.00	**33.20**	**88.93**	39.10	91.61	**150.10**	236.81	154.00	**124.22**	613.30	2428.99	**520.60**	**606.12**	1948.67	21250.37	**1654.10**	**3306.58**
q_query_3_l39	36.93	**23.60**	60.92	30.80	**60.84**	**94.10**	124.92	99.10	**81.75**	341.40	834.12	**329.90**	**441.85**	1048.00	8935.69	**998.70**	**2550.71**
q_query_3_l40	35.53	**25.20**	53.05	27.80	**48.00**	115.40	115.92	**97.90**	**61.46**	334.10	822.02	**289.30**	**394.90**	1000.60	11156.29	**861.50**	**2242.65**
q_query_3_l41	35.85	**14.70**	**49.58**	17.10	51.11	**52.30**	69.37	53.00	**58.76**	158.50	**298.53**	**147.90**	322.85	460.70	3737.47	**432.30**	**1415.41**
q_query_3_l42	32.24	15.90	37.66	**14.80**	**33.63**	58.10	58.10	**48.70**	**38.62**	194.50	262.37	**155.10**	**149.09**	567.70	2773.66	**482.30**	**1168.59**
q_query_3_l43	38.63	**12.40**	47.41	12.60	**43.01**	44.10	65.14	**34.90**	**50.45**	110.10	190.81	**97.00**	**145.47**	286.10	1325.07	**261.20**	**1258.91**
rpoc_xits_17	80.93	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
unconstr	1.12	**14.00**	**4.54**	22.90	4.57	39.20	**1.27**	**38.60**	4.27	**101.10**	**2.10**	108.10	121.21	**263.80**	**21.42**	318.30	519.45

Table 3. Average test suite size and runtime for the instances with 50 parameters and 5000 conflicts

We did not find any state-of-the-art tool capable of generating MCACs that could solve the benchmarks produced with SUT-G. We evaluated the ACTS tool [9] using its *Relation* feature, but ACTS failed to generate a test suite for any benchmark or strength. We tested both the *IPOG* and *IPOF* algorithms, using either a Constraint Programming (CP) solver [21] or Minimum Forbidden Tuples (MFT) to handle constraints. Consequently, ACTS results are not included in the following tables.

To address this limitation, we developed our own IPOG implementation, called *CTLog*, with full support for both the *ACTS* and *Extended ACTS* SUT formats. To our best knowledge, the main difference from ACTS is constraint handling: CTLog uses the Glucose 4.1 SAT solver [7], whereas ACTS relies on a CP solver. Additionally, CTLog is implemented in Python and Nim, while ACTS is implemented in Java. We also observed that differences in tie-breaking strategies significantly affect performance.

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-15	89.81	**36.40**	240.82	45.60	**195.86**	**204.00**	1546.95	221.90	**487.02**	**863.20**	19179.87	911.20	**2205.40**	-	-	**3183.80**	**14430.61**
UR-10	65.80	**7.70**	70.86	8.60	**70.54**	**15.40**	74.72	16.20	**73.18**	**31.30**	141.40	**31.30**	**126.50**	**58.70**	1257.77	59.90	**1173.76**
q_query_3_L60	69.79	**24.80**	**134.98**	28.40	155.10	**99.50**	249.01	116.10	**242.80**	**368.50**	2325.84	408.60	**641.79**	**1131.80**	21617.18	1202.30	**3397.64**
q_query_3_l37	34.63	**21.60**	**99.26**	29.90	138.58	**114.90**	**261.49**	167.00	421.72	**536.30**	1361.00	637.90	**1110.30**	2264.00	22742.85	**2240.30**	**4821.38**
q_query_3_l38	36.49	**31.60**	63.81	37.70	**58.47**	**144.00**	197.35	149.00	**81.50**	564.50	3008.04	**523.90**	**533.48**	1835.33	19568.58	**1765.60**	**3653.99**
q_query_3_l39	35.68	**24.50**	57.63	26.90	**53.53**	103.80	126.52	**97.80**	**72.24**	384.90	1825.69	**326.80**	**313.45**	1213.38	19632.27	**1071.50**	**2465.20**
q_query_3_l40	36.96	**25.70**	**72.97**	32.20	76.32	**98.70**	120.17	110.50	**106.74**	386.50	665.05	**368.10**	**519.08**	1233.50	6747.26	**1159.80**	**2850.64**
q_query_3_l41	38.58	**25.50**	**88.76**	30.90	99.62	**100.50**	134.56	109.20	**119.79**	351.60	655.93	**339.00**	**435.02**	1088.40	6938.15	**1023.20**	**2613.88**
q_query_3_l42	37.70	**18.70**	**73.24**	22.30	91.13	**70.00**	113.32	70.90	**101.29**	233.70	683.15	**209.40**	**390.85**	658.50	10750.48	**610.40**	**1834.67**
q_query_3_l43	40.14	**12.20**	47.73	13.00	**43.77**	37.00	67.32	**36.00**	**50.48**	108.50	389.33	**106.30**	**257.28**	**312.20**	7074.27	315.90	**1349.14**
rpoc_xits_17	102.53	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
unconstr	1.12	**14.00**	**4.54**	22.90	4.57	39.20	**1.27**	**38.60**	4.27	**101.10**	**2.10**	108.10	121.21	**263.80**	**21.42**	318.30	519.45

Table 4. Average test suite size and runtime for the instances with 50 parameters and 10000 conflicts

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-01	43.81	**15.80**	55.88	23.70	**49.75**	**50.00**	90.37	51.80	**89.54**	156.60	**247.81**	**150.60**	542.48	**434.30**	2931.40	438.00	**2868.83**
AProVE09-07	9.57	**11.60**	**14.43**	13.80	16.84	**18.40**	**34.44**	21.00	37.78	**20.00**	1212.73	25.20	**1020.87**	-	-	-	-
AProVE09-08	11.42	**5.00**	**19.37**	6.50	21.36	**5.00**	**30.36**	7.90	39.13	**5.00**	**957.82**	7.90	965.14	-	-	-	-
AProVE09-20	34.10	16.90	**92.54**	**16.70**	109.05	54.80	**132.05**	**52.30**	138.03	152.50	1919.41	**145.80**	**1075.61**	-	-	-	-
AProVE09-21	34.04	**94.90**	**2838.59**	114.30	3628.20	**865.00**	**26293.02**	958.00	27933.50	-	-	-	-	-	-	-	-
gss-14	31.99	**1.00**	**60.37**	2.00	76.75	**1.00**	83.62	2.00	**82.66**	**1.00**	**670.06**	2.00	923.77	**1.00**	**26100.93**	-	-
q_query_3_L60.	71.47	**63.80**	**170.84**	71.30	170.90	**430.70**	2060.01	456.70	**557.46**	-	-	**2193.90**	**5705.75**	-	-	-	-
q_query_3_l37	32.21	**39.90**	61.24	42.50	**50.16**	219.60	676.36	**194.10**	**181.93**	1010.25	18530.40	**828.60**	**2142.49**	-	-	-	-
q_query_3_l38	34.90	**47.80**	89.90	59.00	**77.89**	278.60	1060.78	**258.60**	**249.47**	1293.00	24972.61	**1183.50**	**2851.65**	-	-	-	-
q_query_3_l39	33.07	**46.80**	83.15	59.70	**77.65**	277.10	733.63	**267.10**	**251.56**	1388.33	20305.54	**1167.40**	**2547.66**	-	-	-	-
q_query_3_l40	37.51	**28.60**	64.57	31.70	**52.70**	146.30	441.82	**125.00**	**145.61**	583.62	9742.55	**473.50**	**1335.30**	-	-	**1816.30**	**26353.09**
q_query_3_l41	36.45	16.50	47.81	**16.40**	**40.61**	67.00	107.10	**60.60**	**71.14**	264.30	1123.61	**225.90**	**952.36**	993.50	**28561.22**	**929.67**	28874.10
q_query_3_l42	36.88	**24.50**	67.24	27.20	**63.41**	112.40	197.86	**105.30**	**125.36**	456.70	3437.92	**391.20**	**1225.15**	-	-	**1566.80**	**27627.13**
q_query_3_l43	39.98	**17.80**	72.47	18.20	**61.74**	76.90	250.36	**63.20**	**99.00**	305.00	6928.32	**224.00**	**1102.06**	-	-	**840.80**	**26714.46**
rpoc_xits_17	80.93	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
unconstr	1.13	**16.00**	**4.62**	30.80	4.99	**48.30**	**1.84**	48.80	30.26	**131.40**	**13.21**	145.70	402.53	**342.70**	**977.01**	431.50	1434.26

Table 5. Average test suite size and runtime for the instances with 100 parameters and 5000 conflicts

To validate the competitiveness of CTLog, we compared it against ACTS on 58 standard benchmarks [10, 15, 20, 18] for strengths $t = 2, 3, 4$, which are supported by both tools. We additionally included the *BOT-its* algorithm as a baseline for future comparisons. The results are reported in Table 7 (Appendix). ACTS was executed once, while CTLog was run five times with different random seeds to mitigate stochastic effects[5] We report average test suite sizes and runtimes. CTLog solved 9 more instances than ACTS across all tested strengths. Overall, CTLog generated MCACs faster than ACTS, while achieving comparable test suite sizes. This justifies the use of CTLog as a replacement for ACTS.

Finally, we compared IPOG against the *BOT-its* and *PBOT-its* algorithms [4]. All algorithms are implemented in Python, with performance-critical components implemented in Nim. We use the OptiLog framework [1] together with the Glucose 4.1 SAT solver [7]. Both BOT-its variants were adapted to support the extended SUT format with auxiliary variables. As shown in Table 7 (Appendix), BOT-its generally exhibits worse runtimes than CTLog's IPOG implementation. We analyze how this behavior changes on benchmarks generated with SUT-G in the following section.

[5] ACTS does not expose a seed parameter.

		$t=2$				$t=3$				$t=4$				$t=5$			
		ipog		bot		ipog		bot		ipog		pbot1G		ipog		pbot1G	
	loading	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
AProVE09-07	11.41	**12.90**	**28.05**	15.20	34.36	**23.60**	**61.76**	28.30	67.58	**30.30**	2286.75	40.80	**1061.32**	-	-	-	-
AProVE09-08	9.95	**6.40**	**17.59**	6.70	18.18	**8.40**	34.37	10.30	**33.44**	**9.00**	1246.54	11.30	**1034.52**	-	-	-	-
AProVE09-15	108.00	**79.50**	785.39	95.70	**647.22**	**547.56**	20669.86	614.90	**2721.84**	-	-	**3269.00**	**26733.10**	-	-	-	-
UR-10	79.13	**17.10**	106.90	18.10	**98.14**	80.50	409.07	**57.40**	**123.16**	202.40	9154.27	**166.30**	**1095.66**	-	-	-	-
gss-13	33.57	**1.00**	**62.77**	2.00	69.51	**1.00**	**83.47**	2.00	96.69	**1.00**	**681.71**	2.00	1022.49	**1.00**	**26564.06**	-	-
q_query_3_L60	71.19	**59.20**	**190.63**	69.50	203.05	**480.10**	2087.49	516.10	**657.57**	-	-	**2789.00**	**8397.63**	-	-	-	-
q_query_3_l37	36.41	**41.40**	**162.22**	54.00	187.84	**278.70**	2744.37	282.30	**634.65**	-	-	**1392.70**	**3563.05**	-	-	-	-
q_query_3_l38	35.34	**44.80**	**136.72**	59.50	183.06	277.80	1408.45	**277.50**	**464.04**	-	-	**1268.90**	**3116.38**	-	-	-	-
q_query_3_l39	39.15	**32.60**	**173.30**	41.00	181.19	**182.10**	937.23	190.80	**420.84**	986.00	19424.75	**839.40**	**2314.78**	-	-	-	-
q_query_3_l40	38.41	**36.40**	94.47	43.70	**85.52**	216.40	953.06	**193.70**	**262.57**	1017.80	18365.59	**862.60**	**2100.76**	-	-	-	-
q_query_3_l41	38.12	**34.60**	**82.15**	35.70	96.16	182.10	808.45	**152.80**	**203.69**	800.60	19193.14	**624.50**	**1712.61**	-	-	-	-
q_query_3_l42	41.18	**31.80**	91.80	34.20	**88.67**	169.50	607.81	**148.20**	**226.93**	824.80	15438.05	**633.20**	**1697.92**	-	-	-	-
q_query_3_l43	41.14	**23.30**	**92.72**	24.10	111.16	110.60	457.17	**88.90**	**184.50**	451.67	11020.63	**341.50**	**1316.03**	-	-	**1398.25**	**27632.00**
unconstr	1.13	**16.00**	**4.62**	30.80	4.99	**48.30**	**1.84**	48.80	30.26	**131.40**	**13.21**	145.70	402.53	**342.70**	**977.01**	431.50	1434.26

Table 6. Average test suite size and runtime for the instances with 100 parameters and 10000 conflicts

Algorithm SUT-G also uses Glucose 4.1 [7] as an incremental SAT solver. To introduce variability in the `solve_subproblem` function, we set the solver parameters `rnd-freq` and `rnd-pol-freq` to 0.5. We also set $\Delta_{\mathcal{A}} = 10$, $\nabla_{\mathcal{A}} = 5$, $\mathcal{S}$ to five random seeds, and $MAX\_TRIES = 100$. Each SAT solver call was limited to 300s. If any call exceeded this limit, the instance was treated as UNSAT.

4.1 RQ1: Comparing SUT-G with random SUT generators

We focussed our comparison on *BOOLC* track of the CT Competition 2022[6], which contains SUTs with only boolean parameters and randomly generated constraints. Since benchmarks produced by SUT-G fit these characteristics and *BOOLC* benchmarks were generated using a random generator[7], they could also belong to this track.

The largest *BOOLC* SUT contain 20 parameters. To enable a fair comparison, we discarded benchmakrs with fewer than 10 parameters. We evaluated the IPOG and BOT-its algorithms on all these benchmarks. Table 11 (Appendix) reports results for the *BOOLC* track, while Tables 1 and 2 show results for SUT-G benchmarks with 10 parameters and conflict limits of 5,000 and 10,000, respectively.

For the *BOOLC* benchmarks, all algorithms produced test suites within 14 seconds for strengths $t < 4$. For $t = 5$, the maximum runtime increased to about 40 seconds, observed in some BOT-its instances. As expected, test suite sizes increase with t, but runtimes do not show a corresponding growth, indicating that the constraints in this track are relatively easy for the tested algorithms.

In contrast, SUT-G benchmarks exhibit significantly higher runtimes and greater variability. To rule out bias from formula loading, we report loading times separately in Tables 1 and 2. Even after discounting these costs, SUT-G bench-

[6] CT-Competition: https://fmselab.github.io/ct-competition/

[7] The random benchmark generator is available here: https://github.com/fmselab/CIT_Benchmark_Generator.

marks remain substantially more demanding, showing that their constraints have a stronger impact on MCAC algorithms.

Moreover, runtimes increase significantly with higher strengths. This effect is particularly evident in the *AProVE09-20* benchmark (Table 2), where IPOG's runtime grows from 80 seconds at $t = 2$ to 124 seconds at $t = 5$.

Finally, the relative performance of the algorithms differs across benchmarks. In the *BOOLC* track, IPOG is consistently the fastest as t increases. In contrast, on SUT-G benchmarks, BOT-its outperforms IPOG at higher strengths. Section 4.3 further analyzes this shift, which we attribute to the greater diversity and complexity of constraints generated by SUT-G, features that appear to be underrepresented in existing CT benchmarks and random generators.

4.2 RQ2: Impact of SUT constraints in SUT-G benchmarks

Our second research question examines how SUT constraints affect the performance of the evaluated algorithms. To isolate this effect, we included, for each benchmark, an execution with the same number of parameters but without constraints (reported as the *unconstr* row in Tables 1, 2, 3, 4, 5, and 6).

Overall, adding SUT constraints consistently increases MCAC generation time, independently of the number of parameters, the strength t, or the algorithm. This behaviour is expected, as candidate test cases must be checked for consistency with the constraints. The runtime penalty becomes more pronounced as the number of parameters, the strength, or the *hardness* of the constraints (measured by the number of conflicts) increases.

On the other hand, the impact on test suite size is more variable. In some benchmarks, constraints increase the number of tests (e.g., *AProVE09-21* in Table 3 across all strengths and algorithms), while in others they reduce it (e.g., *UR-10* in Table 4). In the former case, combinations that were previously covered by a single test become forbidden, forcing tests to be split and increasing the suite size. In the latter case, strong constraints eliminate many valid combinations, reducing the number of required tests.

Notice that neither IPOG nor BOT-its guarantee optimal test suite size (i.e., they do not certify the *CAN*), so smaller test suites may exist. Nevertheless, these results offer valuable insight into the behaviour of practical MCAC generation algorithms under *hard* SUT constraints. In contrast, complete approaches that aim to compute optimal MCACs, such as MaxSAT-based MCAC [6, 2] or CALOT [19], are likely to struggle with benchmarks of this complexity.

4.3 RQ3: Comparing the performance of the IPOG and (P)BOT-its algorithms

In Section 4.2, we analyzed the overall impact of constraints by comparing generated benchmarks with their unconstrained counterparts Here, we focus on performance differences between the *IPOG* and *(P)BOT-its* algorithms. As noted

at the beginning of Section 4, we also evaluated four ACTS algorithms. However, none produced a test suite for the generated benchmarks, even for strength $t = 2$, so we omit these results.

Overall, *IPOG* produces smaller test suites than *BOT-its*,especially for $t < 4$, but at the cost of longer runtimes. *BOT-its* incrementally constructs tests that may be temporarily inconsistent and later repairs them to reduce SAT solver calls While this strategy can lead to suboptimal choices and larger test suites, it significantly reduces runtime, particularly for benchmarks with hard SUT constraints. This behavior contrasts with prior results, where *IPOG* was faster than *BOT-its* on state-of-the-art benchmarks [18, 4]. We observe the same trend in Tables 7 and 11 in the Appendix (see also Section 4.1).

For the 100-parameter benchmarks in Tables 5 and 6, both algorithms exhibit memory limitations. When $t > 3$, *IPOG* completes only 25 of 56 experiments, while the basic *BOT-its* fails in all cases.[8] Consequently, we ran the pooled version *PBOT-its* with a 1GB pool (*pbot1G*, see [4]), which successfully solves 32 of the 56 experiments.[9] In these settings, runtime differences between *IPOG* and *PBOT-its* are substantial, up to an order of magnitude for several benchmarks (e.g., *q_query_3_l37-l39* in Table 5 and *q_query_3_l39-l43* in Table 6 for $t = 4$). With the latest *PRBOT-its* variant, we observe that it often outperforms *IPOG* in test suite size, particularly for higher strengths ($t > 3$) and larger models (50 or more parameters).

Finally, one benchmark (*rbcl_xits_17*) remains unsolved by all algorithms. Although Algorithm SUT-G aims to control benchmark hardness, some instances appear to force exploration of exponential branches when completing test cases. *IPOG* cannot avoid this behavior, whereas *BOT-its* could potentially mitigate it by adapting the *amend* procedure, an avenue for future work.

Overall, the *SUT-G* generator provides new insights into the strengths and limitations of established MCAC algorithms. In the next section, we derive practical guidelines for applying these techniques in real-world scenarios.

5 Conclusions and Future Work

We have introduced a new generator for SUT benchmarks. Its simple yet flexible design provides the CT research community with access to a variety of SAT-based structures that reflect real-world problems. In particular, it allows generating SUT benchmarks of specified size and constraint hardness, facilitating the development and evaluation of new CT tools. This approach can also be extended to leverage instances from other constraint programming or operations research formalisms. Additionally, this work presents the first detailed study of existing CT tools, helping to guide tool selection and improve testing strategies for novel SUTs.

We also highlight the critical importance of properly handling SUT constraints in all MCAC algorithms. As noted in Section 1, future real-world CT

[8] These results are omitted from the tables.
[9] The remaining instances exceeded the timeout.

applications are likely to involve constraints similar to those generated by *SUT-G*. Algorithms that cannot efficiently manage these constraints risk limiting the adoption of CT techniques in complex environments.

For future work, we plan to extend *SUT-G* to support mixed-domain SUTs, achievable by using CSP, MIP, or MiniZinc instances (which support finite-domain variables) as input instead of SAT instances. Another approach is to detect CSP variables encoded as SAT variables [5]. A more challenging direction is to analyze how SUT constraints affect the optimal Covering Array size for a given strength, ultimately enabling the generation of SUTs whose minimal MCAC size is predictable within defined bounds.

References

1. Alos, J., Ansótegui, C., Salvia, J.M., Torres, E.: Optilog V2: model, solve, tune and run. In: Meel, K.S., Strichman, O. (eds.) 25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel. LIPIcs, vol. 236, pp. 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.SAT.2022.25, https://doi.org/10.4230/LIPIcs.SAT.2022.25
2. Ansótegui, C., Izquierdo, I., Manyà, F., Torres-Jiménez, J.: A max-sat-based approach to constructing optimal covering arrays. In: Artificial Intelligence Research and Development - Proceedings of the 16th International Conference of the Catalan Association for Artificial Intelligence, Vic, Catalonia, Spain, October 23-25, 2013. pp. 51–59 (2013)
3. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with boolean variables. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings. pp. 1–15 (2004)
4. Ansótegui, C., Ojeda, J., Torres, E.: Building high strength mixed covering arrays with constraints. In: Michel, L.D. (ed.) 27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021. LIPIcs, vol. 210, pp. 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CP.2021.12, https://doi.org/10.4230/LIPIcs.CP.2021.12
5. Ansótegui, C.: Complete SAT solvers for Many-Valued CNF Formulas. Ph.D. thesis, University of Lleida (2004)
6. Ansótegui, C., Manyà, F., Ojeda, J., Salvia, J.M., Torres, E.: Incomplete MaxSAT approaches for combinatorial testing. Journal of Heuristics **28**(4), 377–431 (Aug 2022). https://doi.org/10.1007/s10732-022-09495-3, https://doi.org/10.1007/s10732-022-09495-3
7. Audemard, G., Lagniez, J.M., Simon, L.: Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In: International conference on theory and applications of satisfiability testing. pp. 309–317. Springer (2013)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press (2021). https://doi.org/10.3233/FAIA336, https://doi.org/10.3233/FAIA336

9. Borazjany, M.N., Yu, L., Lei, Y., Kacker, R., Kuhn, R.: Combinatorial testing of ACTS: A case study. In: Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012. pp. 591–600 (2012)
10. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering **34**(5), 633–650 (2008)
11. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4), 1–26 (Jan 2006), publisher: IOS Press
12. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Trans. Software Eng. **30**(6), 418–421 (2004)
13. Nie, C., Leung, H.: A survey of combinatorial testing. ACM Computing Surveys (CSUR) **43**(2), 1–29 (2011)
14. SAT-Competition: (2009), www.satcompetition.org/2009
15. Segall, I., Tzoref-Brill, R., Farchi, E.: Using binary decision diagrams for combinatorial test design. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. p. 254–264. Association for Computing Machinery, New York, NY, USA (2011)
16. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
17. Tzoref-Brill, R., Maoz, S.: Modify, enhance, select: Co-evolution of combinatorial models and test plans. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 235–245. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236024.3236067, https://doi.org/10.1145/3236024.3236067
18. Yamada, A., Biere, A., Artho, C., Kitamura, T., Choi, E.H.: Greedy combinatorial test case generation using unsatisfiable cores. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. p. 614–624. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016)
19. Yamada, A., Kitamura, T., Artho, C., Choi, E., Oiwa, Y., Biere, A.: Optimization of combinatorial testing by incremental SAT solving. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015. pp. 1–10 (2015)
20. Yu, L., Duan, F., Lei, Y., Kacker, R.N., Kuhn, D.R.: Constraint handling in combinatorial test generation using forbidden tuples. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–9 (2015)
21. Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R.N., Kuhn, D.R.: An efficient algorithm for constraint handling in combinatorial test generation. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 242–251 (2013). https://doi.org/10.1109/ICST.2013.35

Appendix 1.A Comparison with state-of-the-art benchmarks

	$t=2$						$t=3$						$t=4$					
	ACTS		ipog		bot		ACTS		ipog		bot		ACTS		ipog		pbot1G	
	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
[10]																		
1	48	9	45.00	7.06	46.00	7.45	293	12	289.40	8.59	289.80	68.41	1680	1207	1671.60	163.59	1662.40	843.20
2	32	9	34.20	7.05	32.80	7.37	174	6	179.60	6.63	164.80	48.46	873	586	869.60	78.77	770.60	527.47
3	19	9	19.80	4.32	18.80	4.36	71	2	66.60	3.80	67.40	4.45	212	2	211.20	1.97	202.80	25.36
4	22	7	22.60	6.56	24.60	6.97	102	3	101.40	4.80	102.20	11.79	374	12	383.00	15.75	379.80	247.18
5	54	10	51.80	8.02	53.00	8.63	386	177	376.20	17.73	377.00	364.77	-	-	2428.20	1147.38	2427.20	1548.06
6	25	8	26.60	6.72	27.60	7.07	119	5	121.00	6.17	124.40	21.33	491	116	494.00	89.00	480.80	417.02
7	12	9	12.20	4.25	13.60	4.32	35	2	35.00	3.60	32.20	4.32	93	2	97.00	2.54	90.80	16.47
8	47	8	45.60	4.98	46.40	5.05	326	63	319.40	10.27	299.60	109.70	1988	6819	1898.20	369.33	1731.80	1055.62
9	22	9	21.20	4.74	21.80	4.62	84	4	83.40	5.02	80.60	11.08	268	77	272.80	45.76	260.60	311.03
10	47	8	46.80	5.50	49.60	5.45	329	117	329.20	16.33	337.80	218.58	2063	17447	2039.40	1183.74	2037.20	1929.33
11	47	8	46.80	7.08	49.00	7.34	318	26	317.00	8.64	323.40	69.26	1885	2150	1897.40	154.72	1910.20	985.23
12	43	11	41.00	7.30	44.80	8.03	263	46	260.00	12.75	261.80	188.68	1465	5676	1430.40	701.64	1386.80	1137.29
13	40	10	38.80	5.09	40.20	4.92	200	10	212.80	8.83	209.60	126.03	1040	1668	1061.60	545.31	972.80	1387.83
14	39	12	39.20	7.22	42.20	7.45	244	4	242.00	7.86	239.60	55.16	1163	225	1163.00	90.85	1186.40	629.79
15	32	9	33.00	4.62	34.40	4.26	173	4	174.40	4.72	173.80	13.93	770	21	774.80	13.77	759.00	323.40
16	25	7	25.60	4.72	27.80	4.59	117	7	119.40	6.78	119.40	29.09	453	321	451.40	140.41	455.60	519.04
17	41	9	40.60	5.18	43.60	5.01	265	48	260.40	9.72	261.40	147.20	1514	4904	1458.20	303.09	1341.80	875.98
18	52	9	47.80	5.58	48.20	5.57	344	27	321.80	12.79	321.60	194.11	2145	3999	1998.00	389.51	1954.80	1042.30
19	51	11	50.00	4.88	54.40	5.66	373	159	374.20	22.98	370.20	560.19	-	-	2488.20	2184.07	2359.60	2333.66
20	60	10	58.40	5.83	60.20	5.68	463	482	464.40	26.13	467.40	320.93	-	-	3261.40	2468.25	3280.80	3318.33
21	39	9	39.80	4.76	42.60	4.66	235	9	233.20	6.29	232.80	32.69	1070	494	1092.20	189.45	1123.40	1232.51
22	37	9	37.00	6.93	36.80	7.33	164	5	166.60	6.38	162.20	26.18	664	100	672.60	73.39	665.60	482.85
23	14	9	14.00	6.52	15.00	6.82	48	2	47.80	3.89	46.60	4.40	140	3	140.60	3.18	132.80	16.68
24	48	8	49.20	5.28	50.00	5.31	341	42	338.20	15.25	332.60	114.61	2105	4512	2096.00	584.83	1981.60	1049.66
25	52	7	53.80	7.27	55.80	7.91	404	58	405.40	13.44	407.60	182.18	2673	9514	2682.80	460.22	2643.00	1293.84
26	34	9	35.20	7.11	36.20	7.40	207	17	208.40	7.53	195.80	51.75	1111	1373	1125.60	154.35	1009.20	675.09
27	37	7	38.20	6.65	39.00	7.19	204	4	215.60	5.42	203.80	17.29	1004	22	1025.60	17.95	965.20	398.05
28	57	8	54.40	7.97	57.20	9.25	420	119	418.60	26.28	421.00	558.22	-	-	2841.80	2394.54	2812.60	1930.58
29	29	7	29.20	4.94	30.00	5.16	154	6	154.60	8.17	149.60	146.24	681	1354	701.60	224.37	660.80	679.48
30	22	7	20.60	4.45	21.60	4.67	100	5	95.80	5.68	89.20	23.20	386	142	383.80	73.63	353.60	405.24
apache	33	9	33.00	5.25	34.40	5.72	173	7	172.60	9.83	181.20	253.13	838	3468	839.60	396.20	848.00	745.01
bugzilla	19	7	18.20	4.39	18.40	4.36	68	2	67.20	3.90	66.20	8.31	242	4	231.00	3.03	221.80	178.88
gcc	23	7	24.20	6.02	22.40	6.61	108	7	192.40	80.83	98.20	349.66	444	3386	548.00	8311.48	416.00	1121.13
spins	26	6	26.00	6.64	25.20	6.82	98	2	114.00	3.79	112.40	4.17	393	3	420.60	2.66	417.80	9.83
spinv	45	7	45.80	6.94	40.60	7.19	286	4	302.40	13.81	236.40	16.75	1631	20	1782.60	707.28	1271.60	545.35
[15]																		
Banking1	15	11	16.40	6.61	15.20	6.91	58	3	55.00	4.28	54.40	4.32	139	3	142.60	1.93	146.80	1.84
Banking2	11	11	11.00	6.63	10.00	6.84	39	2	38.60	4.30	38.80	4.33	96	2	95.20	1.61	94.20	2.34
CommProtocol	19	13	19.80	6.60	19.60	6.90	49	7	50.60	4.51	49.40	4.41	97	36	98.80	1.74	99.80	2.42
Concurrency	6	11	5.60	6.90	5.40	7.16	8	2	8.00	4.22	8.00	4.20	8	2	8.00	1.63	8.00	1.75
Healthcare1	30	10	30.00	6.73	30.00	6.99	105	2	103.20	4.29	105.80	4.32	341	2	324.40	1.61	327.00	2.40
Healthcare2	16	11	16.40	6.83	18.00	7.01	67	2	63.60	4.43	64.80	4.26	220	4	221.00	1.81	217.60	2.76
Healthcare3	38	11	39.00	6.73	36.40	7.00	209	3	203.80	4.82	200.60	6.70	1004	11	974.20	7.39	938.80	154.15
Healthcare4	49	12	48.40	9.66	47.40	9.72	294	3	292.60	5.38	295.40	11.53	1644	15	1630.80	12.26	1607.40	530.12
Insurance	527	11	527.40	7.27	527.00	7.21	6866	2	6932.40	11.25	6862.80	27.35	75764	5	75670.60	99.58	75361.60	4526.85
NetworkMgmt	112	11	116.00	6.76	119.20	6.90	1125	2	1130.00	2.13	1109.00	2.56	6267	3	6212.60	9.14	6152.80	61.48
ProcessorComm1	29	11	28.40	6.54	29.20	6.59	163	2	157.80	5.52	140.80	5.70	670	3	678.40	7.65	614.00	16.18
ProcessorComm2	32	13	30.60	10.21	31.00	10.29	161	65	162.00	5.88	157.20	7.16	744	7250	742.00	15.72	719.20	102.60
Services	106	17	106.60	6.92	102.80	7.38	963	186	915.00	5.67	925.40	6.85	6855	4718	6861.20	12.45	6567.60	190.42
Storage1	17	11	17.60	6.75	18.20	7.15	25	2	25.00	4.38	25.00	4.38	25	2	25.00	1.69	25.00	1.69
Storage2	18	11	18.00	6.76	20.20	6.86	74	1	61.40	4.33	71.80	4.08	195	1	179.80	2.14	200.00	1.72
Storage3	50	12	54.40	12.95	52.60	13.10	239	5	239.60	4.74	238.40	4.83	752	40	755.00	2.58	741.60	17.03
Storage4	136	10	133.80	7.21	130.20	7.42	990	2	1005.40	6.36	964.60	9.05	6636	3	6591.80	20.35	6611.00	551.11
Storage5	218	13	227.80	7.46	236.60	7.46	1879	6	1943.40	9.66	1932.40	25.27	13292	17	13461.60	121.38	13198.40	2436.99
SystemMgmt	17	10	16.60	6.65	18.20	6.95	60	2	55.80	4.44	56.60	4.07	152	2	149.00	1.60	147.20	1.91
Telecom	32	11	30.40	6.61	30.80	6.97	126	2	126.20	4.49	124.20	4.40	392	2	398.20	1.96	401.60	2.88
[20]																		
RL-A	155	436	158.40	7.61	153.40	7.78	-	-	1160.00	12.41	1090.60	34.41	-	-	7761.60	346.16	7306.60	3845.00
RL-B	-	-	760.00	14.64	749.60	13.32	-	-	14107.80	229.95	13482.20	562.12	-	-	-	-	-	-
[18]																		
Company2	81	31	78.20	1.53	78.40	1.48	424	630	363.20	2.04	359.20	2.43	-	-	1391.00	5.90	1380.00	28.20

Table 7. Comparison of IPOG in *ACTS* and IPOG and BOT-its in *CTLog*

Appendix 1.B Features of generated benchmarks

	# vars	# clauses	# lits	Max clause size	Avg clause size
10p-5000c					
AProVE09-01	39556	114648	301635	46	2.630966
AProVE09-05	12080	38902	98687	12	2.53681
AProVE09-07	6960	21780	54683	10	2.510698
AProVE09-08	7068	22197	55774	10	2.512682
AProVE09-21	28095	84176	197133	10	2.341915
gss-13	27797	81345	189313	3	2.327285
gss-14	27183	79047	183529	3	2.321771
q_query_3_L60	40318	170080	438183	11	2.576335
q_query_3_l37	20482	83795	213904	9	2.552706
q_query_3_l38	19772	79783	203135	9	2.546094
q_query_3_l39	15908	67124	173978	9	2.59189
q_query_3_l40	21642	88249	225449	9	2.554692
q_query_3_l41	22269	91036	232764	9	2.556835
q_query_3_l42	22135	89217	227324	9	2.54799
q_query_3_l43	23279	94225	239660	9	2.543486
rpoc_xits_17	2180	132130	384236	18	2.908015
10p-10000c					
AProVE09-05	12153	39174	99423	12	2.537984
AProVE09-07	7052	22100	55513	10	2.5119
AProVE09-15	80896	247518	613147	3	2.477181
AProVE09-20	28282	87376	218173	5	2.496944
UR-10	44159	183277	426366	10	2.326348
q_query_3_L60	41459	175583	453535	11	2.583023
q_query_3_l38	18965	76181	194604	9	2.554495
q_query_3_l39	20924	85056	216661	9	2.547275
q_query_3_l40	21224	85903	219006	9	2.549457
q_query_3_l41	21317	86278	220945	9	2.56085
q_query_3_l42	23097	94482	241230	9	2.553185
q_query_3_l43	22828	93694	240972	9	2.571904
rpoc_xits_17	2469	155422	458659	17	2.951056

Table 8. SUT constraints features for the benchmarks with 10 parameters generated with SUT-G

	# vars	# clauses	# lits	Max clause size	Avg clause size
50p-5000c					
AProVE09-01	39632	114800	302030	46	2.630923
AProVE09-05	13447	44105	112586	12	2.552681
AProVE09-07	7090	22150	55578	10	2.509165
AProVE09-20	28305	87728	219210	5	2.498746
AProVE09-21	28592	86186	202189	10	2.345961
q_query_3_L60	41101	173922	449062	11	2.581974
q_query_3_l37	20088	82023	209623	9	2.555661
q_query_3_l38	21048	86588	221457	9	2.557595
q_query_3_l39	21062	85435	217191	9	2.542178
q_query_3_l40	20907	83820	212825	9	2.539072
q_query_3_l41	20953	84312	215159	9	2.551938
q_query_3_l42	20829	78497	193854	9	2.469572
q_query_3_l43	23132	93108	236464	9	2.539674
rpoc_xits_17	2489	164536	485963	18	2.953536
50p-10000c					
AProVE09-15	77291	232609	572393	3	2.460752
UR-10	42396	170363	393715	10	2.311036
q_query_3_L60	41014	172807	446359	11	2.582991
q_query_3_l37	20102	82297	210251	9	2.554783
q_query_3_l38	20969	85750	218842	9	2.552093
q_query_3_l39	20681	84380	215879	9	2.558414
q_query_3_l40	21398	87443	223613	9	2.557243
q_query_3_l41	22375	91874	235545	9	2.563783
q_query_3_l42	21988	89748	230369	9	2.566843
q_query_3_l43	23399	95504	243924	9	2.554071
rpoc_xits_17	2692	208612	620872	18	2.976205

Table 9. SUT constraints features for the benchmarks with 50 parameters generated with SUT-G

	# vars	# clauses	# lits	Max clause size	Avg clause size
100p-5000c					
AProVE09-01	39600	114683	301752	46	2.631183
AProVE09-07	6863	21475	53897	10	2.509756
AProVE09-08	7908	25693	65340	10	2.543105
AProVE09-20	27813	85313	212238	5	2.487757
AProVE09-21	29114	88226	207320	10	2.349874
gss-14	28237	82820	192899	3	2.329135
q_query_3_L60	41333	173970	448658	11	2.578939
q_query_3_l37	19161	76610	194658	9	2.540895
q_query_3_l38	20384	83067	212034	9	2.552566
q_query_3_l39	20076	80087	203023	9	2.535031
q_query_3_l40	21413	87134	222308	9	2.551335
q_query_3_l41	21472	86188	219718	9	2.549288
q_query_3_l42	20945	86402	222392	9	2.573922
q_query_3_l43	23055	94165	241091	9	2.560304
rpoc_xits_17	2443	163960	481728	18	2.938082
100p-10000c					
AProVE09-07	7034	21992	55204	10	2.510186
AProVE09-08	6621	20201	50233	10	2.486659
AProVE09-15	87111	273544	684768	3	2.503319
UR-10	45852	193479	451671	10	2.33447
gss-13	28027	82243	191569	3	2.329305
q_query_3_L60	39029	165877	428052	10	2.580539
q_query_3_l37	19937	81745	208890	9	2.555386
q_query_3_l38	20441	83146	212716	9	2.558343
q_query_3_l39	21760	89927	230460	9	2.562745
q_query_3_l40	21491	87196	222686	9	2.553856
q_query_3_l41	21233	86346	221450	9	2.564682
q_query_3_l42	23029	94615	242581	9	2.563875
q_query_3_l43	23341	96266	247289	9	2.568809

Table 10. SUT constraints features for the benchmarks with 100 parameters generated with SUT-G

Appendix 1.C BOOLC results

	$t=2$				$t=3$				$t=4$				$t=5$			
	ipog		bot		ipog		bot		ipog		bot		ipog		bot	
	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size	time
0	8.80	2.66	9.20	1.86	22.20	2.25	22.50	2.24	37.20	1.83	37.50	2.15	48.50	1.84	48.20	1.99
1	9.40	7.14	9.70	6.83	22.50	3.55	21.60	3.44	52.10	1.76	50.70	2.02	118.20	2.43	116.30	5.97
2	8.80	4.12	8.80	4.15	19.10	1.63	18.40	1.59	44.50	2.09	42.50	2.39	88.80	5.12	87.70	9.59
3	8.50	7.00	8.20	6.96	20.10	3.59	19.80	3.42	43.30	1.77	42.70	1.82	75.60	2.04	73.60	2.47
4	10.30	4.16	10.20	3.89	31.60	2.58	29.50	2.37	84.80	2.87	80.70	5.12	216.00	3.86	204.90	41.99
5	1.00	13.71	1.00	13.57	1.00	1.95	1.00	1.98	1.00	1.91	1.00	1.95	1.00	3.28	1.00	3.16
6	13.80	7.23	13.00	6.89	30.70	3.85	30.20	3.83	49.10	1.84	49.80	2.16	67.50	2.86	67.10	5.53
7	9.70	7.34	10.00	6.93	24.90	1.76	24.50	1.72	58.90	1.76	58.90	2.14	122.60	3.08	121.30	6.98
8	8.00	6.15	8.20	5.94	17.10	3.53	16.40	3.72	36.10	1.88	36.70	2.00	70.90	1.47	70.50	1.67
9	12.80	6.96	12.60	6.94	36.80	3.56	36.00	3.98	100.50	1.90	95.00	3.66	245.70	4.32	233.30	34.83
10	7.90	7.14	8.00	6.83	18.80	3.59	18.70	3.69	37.80	1.46	37.50	1.60	69.20	1.73	68.20	1.91
12	4.00	3.29	4.00	3.12	4.00	2.57	4.00	2.20	4.00	2.53	4.00	2.10	4.00	2.23	4.00	2.34
13	6.80	4.85	6.90	4.81	9.00	1.91	9.00	1.93	9.00	2.02	9.00	1.94	9.00	1.63	9.00	1.69
15	11.20	7.01	11.20	6.97	32.30	1.88	31.10	1.92	80.70	1.59	79.40	2.23	185.40	2.54	182.40	8.82
16	6.80	3.70	7.00	4.12	9.00	2.32	9.00	2.37	9.00	2.95	9.00	3.16	9.00	5.37	9.00	7.53
17	9.60	3.44	10.30	4.30	24.60	2.49	24.00	2.33	59.10	2.40	58.10	2.47	126.50	2.11	125.30	4.00
18	5.00	11.53	5.00	11.22	6.00	1.84	6.00	1.84	6.00	2.02	6.00	2.03	6.00	4.57	6.00	5.30
19	14.10	7.18	14.00	6.78	40.80	3.68	41.10	3.94	102.30	2.50	101.60	4.45	206.40	4.15	207.40	38.37
20	10.40	4.10	10.70	4.14	28.00	2.61	26.90	2.60	75.20	2.69	70.20	4.03	186.30	5.06	176.20	29.54
21	5.40	4.36	5.40	4.33	8.00	1.71	8.00	1.74	8.00	1.82	8.00	1.77	8.00	2.62	8.00	2.30
22	11.00	6.95	11.90	6.89	30.40	3.58	30.20	3.95	81.60	2.94	76.80	4.70	190.60	4.50	183.70	34.10
25	6.10	2.65	6.10	1.84	10.10	2.12	10.10	2.25	12.00	1.79	12.00	2.01	12.00	2.47	12.00	2.95
26	6.00	7.54	6.10	7.10	10.80	1.73	10.70	1.67	16.00	1.51	16.00	1.53	16.00	1.90	16.00	2.16
28	9.70	4.52	10.00	4.56	24.50	1.66	24.60	1.72	62.90	2.08	62.20	2.42	150.40	2.32	147.10	6.62
30	11.50	7.43	11.00	7.04	37.20	1.85	35.60	1.88	107.50	2.00	102.70	3.71	280.30	3.97	268.00	39.74
31	6.70	6.93	6.00	6.98	10.50	3.61	10.10	3.53	12.00	2.29	12.00	2.45	12.00	2.53	12.00	3.16
33	10.90	7.13	10.10	6.76	30.50	3.61	29.70	3.59	85.70	2.51	78.80	3.63	212.20	4.26	200.20	23.92
34	10.10	4.62	10.30	4.39	27.80	1.69	27.80	1.77	77.70	2.20	73.60	2.85	201.90	2.99	191.30	19.30
36	9.70	6.91	10.00	6.93	26.80	3.62	25.60	3.60	70.80	1.87	69.10	2.53	174.20	2.73	168.30	12.42
38	8.90	3.43	8.60	4.28	22.00	2.71	22.10	2.76	51.20	2.48	51.10	2.68	101.00	2.31	98.10	3.07
40	10.30	4.08	10.70	3.96	29.60	2.13	28.90	2.10	81.40	2.88	77.40	3.56	198.30	3.62	188.50	12.87
41	5.60	7.18	5.40	6.86	8.00	3.54	8.00	3.85	8.00	2.21	8.00	2.17	8.00	4.52	8.00	5.44
42	6.00	3.61	6.00	4.01	6.00	2.46	6.00	2.28	6.00	1.89	6.00	1.78	6.00	1.91	6.00	1.86
43	11.10	6.89	10.60	6.83	32.10	3.62	31.80	3.60	90.80	1.86	88.60	3.94	235.40	2.64	231.30	27.04
44	9.10	3.92	9.30	4.12	21.20	2.14	20.70	2.12	49.20	2.90	47.40	3.30	102.50	4.24	99.70	9.66
45	10.20	2.90	11.00	4.21	25.50	2.90	24.90	2.49	65.40	2.80	62.90	4.20	157.40	4.03	151.60	27.59
47	13.20	4.00	12.20	4.18	40.80	2.66	38.00	2.61	110.40	3.37	104.60	5.00	233.30	5.20	225.80	36.60
48	9.50	4.37	9.90	4.33	25.00	1.70	25.30	1.76	63.10	1.95	63.60	2.47	149.90	3.81	147.30	10.95

Table 11. Results for the IPOG and BOT-its algorithms over the *BOOLC* track of the CT Competition 2022. Benchmarks with less than 10 parameters were discarded.