Building High Strength Mixed Covering Arrays with Constraints

Carlos Ansótegui
Logic & Optimization Group (LOG), University of Lleida, Spain

Jesús Ojeda
Logic & Optimization Group (LOG), University of Lleida, Spain

Eduard Torres
Logic & Optimization Group (LOG), University of Lleida, Spain

Abstract

Covering arrays have become a key piece in Combinatorial Testing. In particular, we focus on the efficient construction of Covering Arrays with Constraints of high strength. SAT solving technology has been proven to be well suited when solving Covering Arrays with Constraints. However, the size of the SAT reformulations rapidly grows up with higher strengths. To this end, we present a new incomplete algorithm that mitigates substantially memory blow-ups. The experimental results confirm the goodness of the approach, opening avenues for new practical applications.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Combinatorial Testing, Covering Arrays, Maximum Satisfiability

Digital Object Identifier 10.4230/LIPIcs.CP.2021.12

Supplementary Material The tools we implemented are available in `http://hardlog.udl.cat/static/doc/prbot-its/html/index.html` as well as detailed installation and execution instructions.

Funding Supported by MICINNs PROOFS (PID2019-109137GB-C21) and FPU fellowship (FPU18/02929).

1 Introduction

Imagine that we want to test a system (a circuit, a program, a cloud application, an industrial engine, a GUI, etc.) to detect errors, bugs, or faults. The System Under Test (SUT) is in essence a black box with a set of input parameters P which take values into a finite domain. These input parameters are assigned to a particular value and then the SUT is run or executed. We assume the only observable output is whether the system crashed or not.

To validate the SUT is working properly, we can simply iteratively conduct a set of tests (assignments of values to the input parameters) and check whether the SUT is working as expected or not. In practice, when the SUT is run, even if we do not explicitly assign a value to a given input parameter it will take its value by default or it will be automatically assigned following some criterion.

Notice that the number of settings (possible tests) to the input parameters (the parameter space) is $\prod_{p \in P} g_p \in \mathcal{O}\left(g^{|P|}\right)$ (where g_p is the cardinality of the domain of parameter p and g is the cardinality of the greatest domain) what yields a *combinatorial* explosion and makes unrealistic to run the SUT under all the possible tests.

Combinatorial Testing (CT) [26] techniques aim to build test suites of a reasonable size but yet powerful enough to *cover* most of the errors, bugs, or faults reported to frequently arise. The point is that, in general, the errors are caused by the interaction of a *relatively small* set of the parameters [22]. Notice that a single test covers $\binom{|P|}{t}$ interactions, where t

(referred to as the *strength*) is the number of parameters involved in the interaction. Therefore, every time we evaluate the SUT under a given test we implicitly check or validate $\binom{|P|}{t}$ interactions of t parameters (referred to as t-tuples).

A test suite of size N for a SUT of P parameters that covers *all* the t-tuples is also known as a Covering Array $CA(N; t, P)$ of strength t. The minimum N for which there exists a $CA(N; t, P)$ is referred to as the Covering Array Number $CAN(t, P)$. Additionally, notice that any test suite of size $< CAN(t, P)$ will not cover all t-tuples, but we may be still interested in covering the maximum number of t-tuples with the number of tests our budget can afford.

In this paper, we show how to build *Mixed* Covering Arrays with *Constraints* (MCACs) of high strength. The term *Mixed* refers to the possibility of having parameter domains of different sizes. The term *Constraints* refers to the existence of some parameter interactions that are not allowed in the system. These forbidden interactions are usually implicitly described by a set of SUT constraints. Therefore, the tests in our test suite must *satisfy* the SUT constraints. In particular, the problem of computing an MCAC of minimum length is NP-hard [24].

There exist several greedy approaches for building MCACs, such as PICT [13] (from Microsoft), based on the OTAT framework [11], and ACTS [10] (used by more than 4000 corporate users and universities), based on the IPOG algorithm [14]. However, they are not well suited in terms of handling SUT constraints and will scale poorly as the complexity or hardness of the SUT constraints grows. This is why, here, we focus on constraint programming based approaches; particularly, we work with Satisfiability (SAT) based approaches [9].

SAT technology provides a highly competitive generic problem approach for solving decision and optimization problems. In particular, the decision problem to be solved is translated to the SAT problem which determines whether there is an assignment to the Boolean variables in a propositional formula in Conjunctive Normal Form (CNF) (set of clauses) that *satisfies* the formula. Additionally, optimization problems can be translated into the Maximum Satisfiability (MaxSAT) problem which is the optimization version of the SAT problem.

The CALOT [30] tool for building MCACs is based on an incremental SAT solving approach which iteratively decreases the upper bound on the size of the test suite, formulating at every iteration as a SAT problem whether there exists an MCAC of size N, till $CAN(t, P)$ is reached. The CALOT approach is extended by recent work in [2] where a MaxSAT formulation based on [4] is proposed allowing the application of the new generation of complete and incomplete MaxSAT solvers [5]. The initial upper bound for these approaches is computed through the application of the ACTS tool.

While these approaches may be efficient enough for testing some SUTs, the size of the SAT or MaxSAT formulas required for building MCACs rapidly grows with the number of tests and size of SUT constraints but mostly with the strength t taken into consideration.

Regarding the number of tests and size of the SUT constraints, the SAT and MaxSAT formulations of the mentioned approaches need to incorporate at least N copies of the SUT constraints where N is the size of the test suite we try to build. In this sense, if the ACTS tool is not able to provide a good enough upper bound then other strategies need to be taken into account since the trivial upper bound, as discussed, can be unaffordable in terms of size. There are approaches like [29] (based on SAT and the domain-dependent PICT heuristic) and [2] (based on MaxSAT) that mitigate this problem by iteratively constructing the test suite, i.e. adding just one single test at a time that aims to maximize the number of interactions covered so far [1]. The addition of one single test guarantees we only deal with one copy of the SUT constraints.

[1] [2] can add more than one test at each iteration.

Regarding the strength t, the size of the SAT/MaxSAT formulas into existing approaches is proportional to the potential number of allowed interactions, i.e. $\mathcal{O}\left(\binom{|P|}{t} \cdot g^t\right)$ where g is the cardinality of the greatest domain. Typical applications use values of $t = 2$ and barely $t = 3$. However, the more complex the SUT is, the higher the probability that faulty or buggy interactions be caused by a larger number of parameters. Therefore, we need to consider higher values like $t = 4$ and $t = 5$, what clearly is a bottleneck for the mentioned SAT or MaxSAT approaches.

Finally, there are other recent Constraint Programming approaches but they focus on $t = 2$ ([17, 18]) or they do not allow SUT constraints ([21]).

In this paper, we show how we can build practical higher strength MCACs through SAT technology without incurring in memory blow-ups. In particular, we first present a new incomplete algorithm named (Refined Build One Test – Incremental Test Suite) RBOT-its, inspired on Algorithm 5 in [29]. RBOT-its builds the MCAC test by test and optimizes (refines) subsets of the incremental test suite built so far by applying a MaxSAT based approach. Then, we present another incomplete algorithm named PRBOT-its (Pool-based Refined Build One Test – Incremental Test Suite) that iteratively builds the MCAC while simultaneously keeping in a memory pool just a fraction of all the possible t-tuples of the SUT fulfilling the memory size requirements.

The paper is structured as follows. In Section 2 we introduce some definitions on Covering Arrays, SAT and MaxSAT. Section 3 shows how the $CAN(t, P)$ problem can be encoded to MaxSAT. Section 4 presents the BOT-its algorithm (Build One Test – Iterative Test Suite), an algorithm that incrementally builds MCACs test by test. Section 5 presents the RBOT-its algorithm that uses a MaxSAT approach to improve the BOT-its algorithm. Section 6 describes the PRBOT-its algorithm that shows how to adapt RBOT-its to operate on low memory requirements. In Section 7 we study how these approaches compare to the ACTS tool. Finally, in Section 8 we conclude and mention some future work.

2 Preliminaries

We introduce some definitions related to Covering Arrays and SAT technology.

▶ **Definition 1.** *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.

▶ **Definition 2.** *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.*

▶ **Definition 3.** *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 $\mathcal{T}_a = \{\tau \mid \tau \not\models \neg S_\varphi\}$.*

▶ **Definition 4.** *A test case v* covers *a value tuple τ if both assign the same domain value to the variables in the value tuple, i.e., $v \models \tau$. A test suite Υ* covers *a value tuple τ (i.e., $\tau \subseteq \Upsilon$) if there exist a test case $v \in \Upsilon$ s.t. $v \models \tau$. We refer to $v \not\models \tau$ ($\tau \not\subseteq \Upsilon$) when a test case (test suite)* does not cover *τ.*

▶ **Definition 5.** *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* [2]*.*

▶ **Definition 6.** *The MCAC problem is to find an MCAC of size* N*.*

▶ **Definition 7.** *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* $CAN(t, S)$*.*

▶ **Definition 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) is a conjunction of clauses.*

▶ **Definition 9.** *A weighted clause is a pair* (c, w)*, where* c *is a clause and* w*, its weight, is a natural number or infinity. A clause is hard if its weight is infinity (or no weight is given); otherwise, it is soft. A Weighted Partial MaxSAT instance is a multiset of weighted clauses.*

▶ **Definition 10.** *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*.*

▶ **Definition 11.** *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. The cost of a clause* (c, w) *under* I *is 0 if* I *satisfies the clause; otherwise, it is* w*. 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.*

▶ **Definition 12.** *The cost of a formula* ϕ *under a truth assignment* I*, denoted by* $cost(I, \phi)$*, is the aggregated cost of all its clauses under* I*.*

▶ **Definition 13.** *The Weighted Partial MaxSAT problem for an instance* ϕ *is to find an assignment in which the sum of weights of the falsified soft clauses is minimal, denoted by cost(*ϕ*), and all the hard clauses are satisfied. The Partial MaxSAT problem is the Weighted Partial MaxSAT problem where all weights of soft clauses are equal. The SAT problem is the Partial MaxSAT problem when there are no soft clauses. An instance of Weighted Partial MaxSAT, or any of its variants, is unsatisfiable if its optimal cost is* ∞*. A SAT instance* ϕ *is satisfiable if there is a truth assignment* I*, called model, such that* $cost(I, \phi) = 0$*.*

▶ **Definition 14.** *An* Exactly-One *(EO) constraint is a cardinality constraint of the form* $\sum_{i=1}^{n} l_i = 1$ *where* l_i *are propositional literals.*

3 The $CAN(t, S)$ problem as MaxSAT

In this section, we first show a SAT encoding for the MCAC problem inspired on previous approaches [19, 20, 6, 25, 4, 30, 2]. Then, we present the MaxSAT encoding for the $CAN(t, S)$ problem presented in [4, 2]. Exactly One cardinality constraints are translated into CNF through the regular encoding [1, 16].

[2] Notice that the CSPLib 045 problem definition of Covering Arrays [28] does not consider SUT *Constraints*.

First, we encode through variables $x_{i,p,v}$ that a test case i assigns value v to parameter p. We restrict each parameter to take one value per test case as follows (where $[N] = \{1, \ldots, N\}$):

$$\bigwedge_{i\in[N]} \bigwedge_{p\in S_P} \sum_{v\in d(p)} x_{i,p,v} = 1 \qquad (X)$$

In order to enforce the SUT constraints, we convert φ to SAT[3] by substituting each (p, v) in φ by the corresponding literal on the propositional variable $x_{i,p,v}$ for each test case i.

$$\bigwedge_{i\in[N]} CNF\left(S_\varphi\left\{\frac{\neg x_{i,p,v}}{p \neq v}, \frac{x_{i,p,v}}{p = v}\right\}\right) \qquad (SUTX)$$

Variables c_τ^i represent that t-tuple τ is covered by test case i or by any lower test case j, where $1 \leq j \leq i$ (equation $CCX(a)$). To ensure that τ will be covered by some test, we set c_τ^N to be True and c_τ^0 to be False (equations $CCX(b)$ and $CCX(c)$). Notice that only t-tuples that can be covered by a test case are encoded, i.e., $\tau \in \mathcal{T}_a$.

$$\bigwedge_{i\in[N]} \bigwedge_{\tau\in\mathcal{T}_a} \bigwedge_{(p,v)\in\tau} (c_\tau^i \to c_\tau^{i-1} \vee x_{i,p,v}) \qquad \text{(a) } (CCX)$$

$$\bigwedge_{\tau\in\mathcal{T}_a} c_\tau^N \qquad \text{(b)}$$

$$\bigwedge_{\tau\in\mathcal{T}_a} (c_\tau^N \to \neg c_\tau^0) \qquad \text{(c)}$$

▶ **Proposition 15.** *Let $Sat_{CCX}^{N,t,S}$ be $X \wedge SUTX \wedge SCCX$. $Sat_{CCX}^{N,t,S}$ is satisfiable iff a $CA(N; t, S)$ exists.*

As we can see, sets $SUTX$ and CCX will be responsible for memory blow-ups when dealing with a large number of tests or allowed t-tuples.

The presented $Sat_{CCX}^{N,t,S}$ encoding requires an upper bound on N and a way to avoid encoding the forbidden t-tuples. These can be extracted from any suboptimal MCAC solution. We can take as upper bound N the number of tests of the solution and discard all the missing t-tuples (as these will be forbidden). After that, row symmetry breaking techniques can be applied. We can compute which is the parameter tuple of length t with the maximum number r of t-tuples, and then fix these r t-tuples in the first r test cases. Notice that these t-tuples are mutually exclusive and must be covered into different test cases. We will refer to the lower bound as $lb = r - 1$ (i.e. it is not possible to find an MCAC with $r - 1$ tests).

This $Sat_{CCX}^{N,t,S}$ encoding can be extended to a MaxSAT encoding for the $CAN(t, S)$ problem, as described in [2, 4]. We will use an indicator variable u_i that is True iff test case i is part of the MCAC. The objective function of the optimization problem, which aims to minimize the number of variables u_i set to True, is encoded into Partial MaxSAT by adding the following set of soft clauses:

$$\bigwedge_{i\in[lb+2\ldots N]} (\neg u_i, 1) \qquad (SoftU)$$

[3] We consider that φ is already in CNF format.

Notice that we only need to use $N - (lb + 1)$ indicator variables since we know that the covering array will have at least $lb + 1$ tests. To avoid symmetries, it is also enforced that if test case $i + 1$ belongs to the MCAC, so does the previous test case i:

$$\bigwedge_{i \in [lb+2 \ldots N-1]} (u_{i+1} \rightarrow u_i) \qquad (BSU)$$

Finally, we just need to state how variables u_i are related to variables c_τ^i. This constraint reflects that if u_i is False (i.e., tests $\geq i$ are not in the solution), then the tuple τ has to be covered at some test below i:

$$\bigwedge_{i \in [lb+2 \ldots N]} \bigwedge_{\tau \in \mathcal{T}_a} (\neg u_i \rightarrow c_\tau^{i-1}) \qquad (CCU)$$

▶ **Proposition 16.** *Let* $PMSat_{CCX}^{N,t,S,lb}$ *be* $SoftU \wedge BSU \wedge CCU \wedge Sat_{CCX}^{N,t,S}$. *If* $N \geq CAN(t, S)$, *the optimal cost of the Partial MaxSAT instance* $PMSat_{CCX}^{N,t,S,lb}$ *is* $CAN(t, S) - (lb + 1)$, *otherwise it is* ∞.

The main problem with these SAT and MaxSAT encodings is that their size dramatically grows with the number of tests and t-tuples to cover. This makes the SAT-based solving approach unpractical in real scenarios. In the next sections, we show how to avoid memory blow-ups by describing new incomplete approaches.

4 Incremental Test Construction

To reduce the number of tests that we need to encode, the idea is to incrementally build the test suite, test by test. Therefore, at any iteration, we just encode the SUT constraints once.

Algorithm BOT-its (Build One Test - Iterative Test Suite), which is inspired on Algorithm 5 in [29], builds an MCAC by iteratively calling the BuildOneTest (BOT) algorithm (an algorithm that greedily builds a new test, see details below). BOT-its keeps a pool p of the t-tuples yet to cover. Then, it incrementally extends the working test suite Υ by appending the new test v computed by the BOT algorithm. The pool p is simplified by erasing those t-tuples covered by v. Finally, the algorithm returns when the pool becomes empty.

Algorithm BOT-its Build One Test – Incremental Test Suite algorithm.

Input : SUT model S, strength t, consistency check conflict budget cb
Output : Test suite Υ

$\Upsilon \leftarrow \emptyset$ `# Working test suite`
$p \leftarrow$ pool with all t-tuples of S
$sat \leftarrow$ incremental SAT solver initialized with X and $SUTX$ constraints
while $p \neq \emptyset$ **do**
  $v, p \leftarrow BOT(S, p, sat, cb)$
  $\Upsilon \leftarrow \Upsilon \cup \{v\}$
  $p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$ `# Tuples in` p `covered by` v
  $p \leftarrow p \setminus p_v$
return Υ

Next, we show the pseudocode for the BuildOneTest (BOT) algorithm, also inspired on Algorithm 5 in [29]. The BOT algorithm receives the pool p with the t-tuples yet to cover. In order to build the current test, BOT uses the PICT heuristic [13] to identify the parameter tuple (to which we refer as the PICT t-tuple) with most t-tuples in the pool. Then, it selects one to initialize the test under construction (line 1).

Algorithm BuildOneTest (BOT) Inspired on Algorithm 5 in [29].

Input : SUT model S, Tuples pool p, SAT solver *sat*, consistency check conflict budget *cb*
Output: A new test case v
`# All functions can access` S`,` p `and` *sat*
1 $v \leftarrow$ choose $\tau \in p$ as in PICT s.t. $consistent(\tau, \infty)$ `#` v `covers at least` τ
2 **while** *there exist* (p, v) *s.t.* $v \cup \{(p, v)\}$ *covers a tuple in* p **and** $consistent(v \cup \{(p, v)\}, cb)$ **do**
3 Choose such best (p, v) `#` $v \cup \{(p, v)\}$ `covers more tuples in` p
4 $v \leftarrow v \cup \{(p, v)\}$
5 **if** *exists* $\tau \in p$ *s.t.* τ *can be covered in* v **and** $consistent(\tau, cb)$ **then**
6 choose $\tau \in p$ as in PICT
7 $v \leftarrow v \cup \tau$
8 **go to** *line 2*
9 $v \leftarrow amend(v)$
10 **return** v, p

To make sure the PICT selection is consistent with the SUT constraints, BOT runs a consistency check (of unlimited *cb* conflicts). In particular, in function *consistent* in BOT auxiliary functions, a SAT solver is used to check the validity of the parameters assigned so far with respect to the SUT constraints. The SAT instance represents the SUT constraints and the SAT solver is executed using as assumptions the partial assignment of all the fixed parameters in the current test. If the check fails, an unsatisfiable core is retrieved[4], i.e., a subset of the formula that is already unsatisfiable. In particular, the core contains the set of assumptions responsible for the unsat answer. Moreover, the t-tuples in the pool subsumed by the core are removed since these are forbidden tuples (line 4 in function *consistent*). Notice that this way a lazy removal of forbidden tuples is implemented.

After the PICT selection, it iteratively selects from the set of unassigned parameters, the pair parameter-value *(p, v)* that, in combination with the parameters fixed so far, covers at least one t-tuple in the pool, preferring the one that covers most (lines 2 - 4). To preemptively detect if the selected parameter plus the previous partial assignment is inconsistent with the SUT constraints it calls function *consistent* but with a limited number of conflicts *cb*, since the check can be expensive and we can not afford a full check at this point.

Whenever the above process saturates, i.e. reaches a fixpoint, and there are yet unassigned parameters, a new t-tuple is selected as in PICT and assigned to the test. Then, the process starts again (line 8). In this case, we also guarantee the selected tuple is consistent with the SUT constraints running *consistent* function with limited conflicts budget *cb*.

At this point, we have heuristically built a partial test that aims to cover most of the t-tuples in the pool, but we may not be able to extend it to a full test consistent with the SUT constraints. Therefore, the partial test may have to be amended (line 9).

[4] When $cb \neq \infty$ the result of the check might be unknown.

Algorithm BOT auxiliary functions Auxiliary functions for algorithm BOT.

```
   # All functions can access S, p and sat
1  function consistent(τ, cb)
2      if sat.solve(τ, cb) = True then return True
3      else
4          p ← p \ {τ | sat.core() ⊆ τ ∧ τ ∈ p}          # p updated in place
5          return False
6  function amend(υ)
7      while not consistent(υ, ∞) do
8          (p, v) ← most recently fixed (p, v) in υ s.t. (p, v) ∈ sat.core()
9          υ ← υ \ {(p, v)}
10     Fix unfixed parameters in υ according to sat.model()
11     return υ
```

This *amend* process (see BOT auxiliary functions) tries to preserve the greatest slice of the partial test that can be extended to a full test consistent with the SUT constraints through the call to function *consistent* with an unlimited budget. In case the partial test is inconsistent, to amend it, the assumptions in the core are removed in reverse chronological order (lines 7 - 9 in function *amend*) till the SAT solver is able to complete the test satisfying the SUT constraints (line 10).

When the BOT algorithm ends, it returns the new test just built v and the input pool p without those forbidden t-tuples that were detected (line 4 in function *consistent*).

The implementation of Algorithm 5 in [29], on which BOT-its and BOT algorithms are inspired, is not available after request to the authors for reproducibility purposes. Our BOT algorithm, apart from implementation details, differs fundamentally on function *consistent*. In particular, on how we specifically conduct a consistency check with a limited number of conflicts.

5 Refining Test Suites

In Section 4 we showed how algorithm BOT-its builds incrementally an MCAC. Notice that the MCAC might not be optimal (i.e. it may exist a smaller MCAC) since BOT-its is a *greedy* algorithm.

Taking as upper bound the size of the suboptimal MCAC provided by the BOT-its algorithm (see Section 4) we can always try to find an smaller MCAC as described in Section 3. Notice that depending on the number of parameters, the strength t and the number of tests, the Partial MaxSAT encoding might be unreasonably large.

To circumvent this issue we essentially compute whether a portion of the MCAC under construction can be *refined* to use fewer tests but cover the same t-tuples in the pool p. We refer to this portion (test suite) as the *window* to be *refined*.

In this section we present algorithm RBOT-its, which is an improvement over BOT-its. Red lines show the extensions.

In particular, we keep an *sliding window* of tests that starts at $w.i$ and ends in the last test of Υ. This window also keeps track of the t-tuples ($w.p$) of the pool p covered by the window (line 11).

Algorithm RBOT-its Refined BOT-its algorithm. Differences with BOT-its in red.

Input : SUT model S, strength t, consistency check conflict budget cb
Output: Test suite Υ

```
1  Υ ← ∅                                                  # Working test suite
2  p ← pool with all t-tuples of S
3  sat ← incremental SAT solver initialized with X and SUTX constraints
4  w.p ← ∅                                                # Window of covered tuples
5  w.i ← 0                                                # Window starting test index
6  while p ≠ ∅ do
7      υ, p ← BOT(S, p, sat, cb)
8      Υ ← Υ ∪ {υ}
9      p_υ ← {τ | τ ∈ p ∧ υ ⊨ τ}                          # Tuples in p covered by υ
10     p ← p \ p_υ
11     w.p ← w.p ∪ p_υ
12     while window_is_full(Υ, w) do
13         Υ, p, w ← refine(Υ, p, w)
14 Υ, p, w ← refine(Υ, p, w)
15 return Υ
```

We keep track of the potential memory size of the Partial MaxSAT required to refine the window. While we hit the maximum allowed size by our system (i.e. function *window_is_full* in line 12 returns true) we execute the refining process (line 13). As we will see below, the refine process, even reducing the number of tests, it may cause to cover additional t-tuples that were not previously in the window. The side effect is that the window may remain full in terms of memory requirements.

Once the algorithm has covered all t-tuples in p, we apply a last refinement to the last window to ensure that it is refined even if the window is not full (line 14).

Function *refine* in Refine tries to cover the same tuples covered in the window $w.p$ but using less tests. First, it encodes as Partial MaxSAT the problem of building a test suite with the minimum number of tests that covers the t-tuples in the window. This can be achieved by making use of the Partial MaxSAT encoding for the $CAN(t, S)$ problem described in Section 3, but taking as $\mathcal{T}_a$ the set of t-tuples into the window and as upper bound ub the window size.

Then, we run a MaxSAT solver and extract the test suite induced by the solution it reports. If the size of this test suite is smaller than the window size, we use it to replace the window in Υ (line 5). We also update the t-tuples covered by the window, since we may cover extra tuples p_x with the new tests (lines 6 - 8). Otherwise, we reduce the size of the window by excluding the test $w.i$ and update properly the window (lines 10 - 13).

6 Incremental Pool of t-tuples

There is yet a main practical problem with the BOT-its algorithm which is the high memory consumption by the pool of t-tuples to be covered. In particular, when t or the number of parameters is high enough.

In this section we present algorithm PRBOT-its, an extension of RBOT-its (see Section 5) to avoid memory blow-ups by limiting the number of t-tuples to be considered when building a test. Red lines show the differences respect to algorithm RBOT-its.

Algorithm Refine Test suites refinement function.

```
   # refine function can access S, t and b
 1 function refine(ϒ, p, w)
 2     φ ← encode(S, ϒ≥w.i, w.p)
 3     ϒr ← solve(φ)
 4     if ϒr ≠ ∅ and |ϒr| < |ϒ≥w.i| then
 5         Replace ϒ≥w.i by ϒr in ϒ
 6         px ← {τ | τ ∈ p ∧ ϒr ⊨ τ}
 7         p ← p \ px
 8         w.p ← w.p ∪ px
 9     else
10         υr ← test case with index w.i in ϒ
11         ϒ ← ϒ \ {υr}
12         w.p ← w.p \ {τ | τ ∈ w.p ∧ υr ⊨ τ}
13         w.i ← w.i + 1
14     return ϒ, p, w
```

This algorithm works on a partial pool p of size at most b. The pool is incrementally filled with new pending t-tuples, to finally traverse all the t-tuples (line 8). Once the pool p is full, the BOT algorithm is called to build a test that tries to cover as much t-tuples as possible in p (line 9, see Section 4). Then, the algorithm proceeds as algorithm RBOT-its (lines 10 – 16). The main loop ends when the pool is empty and there are not pending tuples (unseen tuples) to add to the pool (function *unseen_tuples?*). Finally, as in algorithm RBOT-its we perform a last refinement.

BOT algorithm has been also modified in the following way. In particular, within function *consistent* (called by BOT algorithm) whenever we discard forbidden tuples, we additionally call function *fill_pool* after line 4 in BOT auxiliary functions, as follows:

$$\Upsilon, p, w, \tau \leftarrow fill\_pool(\Upsilon, p, w, \tau)$$

The goal is to take advantage of the available extra space in the pool thanks to the lazy detection and removal of forbidden tuples. Consequently, the call to function BOT in algorithm PRBOT-its (line 9) is extended with the additional entry parameters Υ, w and output parameters w, τ.

To fill the pool of t-tuples we call function *fill_pool* in Fill pool. This function iteratively adds new t-tuples to the pool that are neither in Υ nor in the pool, till p is full or all t-tuples have been processed (seen) (lines 2 – 4).

New t-tuples are selected taking into account the latest tuple seen τ by calling function *next_tuple* (a total order is implicitly assumed, line 3). Notice that whether τ is a forbidden tuple (not consistent with the SUT constraints) it is handled by the BOT algorithm into the *consistent* function as previously described.

If τ was not already covered in Υ it is added to the pool p. Otherwise, if the new tuple is in particular covered by the current window it is consequently added to the window pool (line 6). Since the window may get full, as in previous algorithms we refine the window pool till it is not full anymore (lines 7 – 8).

Algorithm PRBOT-its Pool-based RBOT-its algorithm. Differences with RBOT-its in red.

Input : SUT model S, strength t, consistency check conflict budget cb, pool budget b
Output : Test suite Υ

```
# All functions can access S, t and b
```
$\Upsilon \leftarrow \emptyset$ # Working test suite
$sat \leftarrow$ incremental SAT solver initilized with X and $SUTX$ constraints
$w.p \leftarrow \emptyset$ # Window of covered tuples
$w.i \leftarrow 0$ # Window starting test index
$p \leftarrow \emptyset$ # Working pool of tuples to cover
$\tau \leftarrow \emptyset$
while $p \neq \emptyset$ **or** $unseen\_tuples?(S, t, \tau)$ **do**
  $\Upsilon, p, w, \tau \leftarrow fill\_pool(\Upsilon, p, w, \tau)$
  $\upsilon, p, w, \tau \leftarrow BOT(S, p, sat, cb, \Upsilon, w)$
  $\Upsilon \leftarrow \Upsilon \cup \{\upsilon\}$
  $p_\upsilon \leftarrow \{\tau \mid \tau \in p \wedge \upsilon \models \tau\}$ # Tuples in p covered by υ
  $p \leftarrow p \setminus p_\upsilon$
  $w.p \leftarrow w.p \cup p_\upsilon$
  while $window\_is\_full(\Upsilon, w)$ **do**
    $\Upsilon, p, w \leftarrow refine(\Upsilon, p, w)$
$\Upsilon, p, w \leftarrow refine(\Upsilon, p, w)$
return Υ

7 Experimental Results

In this section, we report the experimental investigation we conducted to assess the performance of the approaches proposed in the preceding sections. We use a total of 58 SUT instances, which are extracted from [12], with 5 real-world and 30 artificially generated covering array problems, [27] with 20 real-world instances, [31] with two industrial instances and, [29] with another industrial instance.

In Table 1 we show the information about each SUT instance. S_P provides the number of parameters and their domain (e.g. in instance *Banking1*, $3^4 4^1$ means 4 parameters of domain 3 and 1 of domain 4) and, S_φ the number of SUT constraints and their sizes (e.g. instance *Banking1* has 112 constraints that involve 5 parameters, 5^{112} in the table).

The environment of execution consists of a computer cluster with machines equipped with two Intel Xeon Silver 4110 (octa-core processors at 2.1GHz, 11MB cache memory) and 96GB DDR4 main memory. All the experiments were executed with a timeout of 12h and a limit of 12GB of RAM. We executed all the algorithms with 10 different seeds, except for the ACTS tool (as it does not expose the *seed* parameter).

We use Python as a programming language and the Python framework OptiLog [3] that provides bindings to state-of-the-art SAT solvers. For our experimentation, we use Glucose 4.1.

Algorithm Fill pool Fill pool function.

```
   # fill_pool function can access S, t and b
1  function fill_pool(Υ, p, w, τ)
2      while |p| < b and unseen_tuples?(S, t, τ) do
3          τ ← next_tuple(S, t, τ)
4          if τ ⊈ Υ then p ← p ∪ {τ}
5          elif τ ⊆ Υ_{≥w.i} then
6              w.p ← w.p ∪ {τ}
7              while window_is_full(Υ, w) do
8                  Υ, p, w ← refine(Υ, p, w)
9      return Υ, p, w, τ
```

We implemented our own version of BOT-its, as the implementation of Algorithm 5 described in [29] was not available from authors for reproducibility purposes[5]. We also found that our implementation is not able to reproduce exactly the results reported in the original work. In particular, we notice that in our case the sizes of the reported MCACs are just slightly higher. Moreover, our implementation also seems to be significantly slower[6]. Notice the authors used as underlying SAT solver lingeling [7] and we use Glucose 4.1, and this may explain part of the divergence. However, this also means that if the implementation of Algorithm 5 from [29] was available we could probably even get better results with our algorithms RBOT-its and PRBOT-its which extend BOT-its. We set the consistency check conflict budget *cb* parameter for all the BOT-its algorithms to 1 (see Section 4).

For the Refine function in algorithms RBOT-its and PRBOT-its we consider the encoding $PMSat_{CCX}^{N,t,S,lb}$ described in Section 3. We use a custom implementation of the *linear* [15, 23] MaxSAT algorithm that is able to report suboptimal solutions[7], using CaDiCaL as the underlying SAT solver [8]. We set a window size of approximately 500MB, a total time limit for the MaxSAT solver of 180s, and a timeout of 30s between solutions (see Section 5). Notice that this setting could be fine-tuned although we did not carry out this analysis. In previous approaches results are provided up to $t = 3$, here we carry out our experiments for $t = 3$, $t = 4$, and $t = 5$ which, as mentioned previously, are also of interest to many applications.

The first question we address is the impact of RBOT-its, the refined version of BOT-its, in terms of size of the reported test suite and run time for $t = 3$. Moreover, we compare with ACTS. We describe the results in Table 1 under columns *tests* and *time*, respectively. Since all approaches are incremental construction methods, we report (under columns "%") a lower bound on the percentage of allowed t-tuples covered by the retrieved test suite. When the percentage is 100 it means it was possible to build an MCAC. On the other hand, instances that have a "-" in all columns were not able to report any test suite. As we can see, RBOT-its is able to report better MCAC sizes than ACTS and BOT-its on 42 of the 58 instances. This confirms the goodness of the refined approach.

The second question we address is about how much memory is consumed by the BOT-its algorithm. In particular, we estimate the required memory to keep all the t-tuples in memory at the same time. We consider integers of 32 bits and we exclude the memory resources

[5] The tools we implemented are available in `http://hardlog.udl.cat/static/doc/prbot-its/html/index.html` as well as detailed installation and execution instructions.
[6] In [29] their algorithms are implemented in C programming language
[7] Since RBOT-its is incomplete by nature, there is actually no need to use a complete MaxSAT solver.

Table 1 SUT parameters domains and constraints for each instance (columns S_P and S_φ) and memory consumption for $t = 3$ (*mem*). Test suite size, percentage of tuple coverage and time for $t = 3$. In bold the method with better results with the lexicographic criteria (coverage percentage, number of tests, exhausted time). For the coverage percentage enough precision was taken into account. Resources: 12GB memory and 12h timeout.

	S_P	S_φ	mem	$t=3$ ACTS			BOT-its			RBOT-its		
inst				tests	%	time	tests	%	time	tests	%	time
Cohen et al. [12]												
1	$2^{86}3^34^15^16^2$	$2^{20}3^34^1$	20.1MB	**293**	**100%**	**4s**	294.20	100%	12m	294.20	100%	1.3h
2	$2^{86}3^34^35^16^1$	$2^{19}3^3$	15.6MB	174	100%	3s	176.50	100%	6m	**149.10**	**100%**	**39m**
3	$2^{27}4^2$	2^93^1	416.0kB	71	100%	1s	72.90	100%	4s	**50.50**	**100%**	**5m**
4	$2^{51}3^44^25^1$	$2^{15}3^2$	3.7MB	102	100%	2s	108.10	100%	48s	**81.10**	**100%**	**7m**
5	$2^{155}3^74^35^56^4$	$2^{32}3^64^1$	112.7MB	386	100%	14s	**384**	**100%**	**1.6h**	384	100%	3.3h
6	$2^{73}4^36^1$	$2^{26}3^4$	8.1MB	119	100%	2s	133.20	100%	2m	**98.60**	**100%**	**14m**
7	$2^{29}3^1$	$2^{13}3^2$	399.7kB	35	100%	1s	39	100%	3s	**28.40**	**100%**	**3m**
8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	34.5MB	326	100%	5s	306.60	100%	23m	**306.20**	**100%**	**1.1h**
9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	4.2MB	84	100%	2s	94.30	100%	44s	**60**	**100%**	**4m**
10	$2^{130}3^64^55^26^4$	$2^{40}3^7$	68.2MB	**329**	**100%**	**9s**	342.60	100%	51m	341.30	100%	2.4h
11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	20.1MB	**318**	**100%**	**4s**	328.70	100%	13m	328.60	100%	1.4h
12	$2^{136}3^44^35^16^3$	$2^{23}3^4$	60.5MB	263	100%	7s	269.80	100%	36m	**250**	**100%**	**1.6h**
13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	43.5MB	200	100%	7s	214.40	100%	19m	**183.70**	**100%**	**1.0h**
14	$2^{81}3^54^36^3$	$2^{13}3^2$	16.3MB	244	100%	3s	244.30	100%	7m	**216.30**	**100%**	**20m**
15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	4.1MB	173	100%	2s	180.10	100%	1m	**150.90**	**100%**	**5m**
16	$2^{81}3^34^26^1$	$2^{30}3^4$	11.6MB	117	100%	3s	138.50	100%	3m	**96.40**	**100%**	**9m**
17	$2^{128}3^34^25^16^3$	$2^{25}3^4$	48.3MB	265	100%	6s	263.50	100%	30m	**239.40**	**100%**	**1.3h**
18	$2^{127}3^24^56^2$	$2^{23}3^44^1$	59.9MB	344	100%	8s	**327.20**	**100%**	**41m**	327.20	100%	2.1h
19	$2^{172}3^94^35^36^4$	$2^{38}3^5$	166.3MB	373	100%	21s	385	100%	2.6h	**365.50**	**100%**	**6.7h**
20	$2^{138}3^44^35^46^7$	$2^{42}3^6$	94.5MB	**463**	**100%**	**12s**	465.60	100%	1.5h	465.60	100%	4.3h
21	$2^{76}3^34^25^16^3$	$2^{40}3^6$	13MB	235	100%	3s	235.40	100%	5m	**216.50**	**100%**	**17m**
22	$2^{72}3^44^16^2$	$2^{20}3^2$	9.3MB	164	100%	2s	164.70	100%	3m	**144**	**100%**	**8m**
23	$2^{25}3^16^1$	$2^{13}3^2$	352.7kB	48	100%	1s	55.40	100%	3s	**37.30**	**100%**	**3m**
24	$2^{110}3^25^36^4$	$2^{25}3^4$	34.5MB	341	100%	5s	**337.70**	**100%**	**25m**	337.70	100%	1.6h
25	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	54.3MB	**404**	**100%**	**7s**	407.70	100%	47m	407.70	100%	2.6h
26	$2^{87}3^14^35^4$	$2^{28}3^4$	16.8MB	207	100%	3s	205.10	100%	7m	**195.30**	**100%**	**47m**
27	$2^{55}3^24^25^16^2$	$2^{17}3^3$	5.1MB	204	100%	2s	210.90	100%	2m	**180.50**	**100%**	**10m**
28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	160.7MB	**420**	**100%**	**21s**	421.80	100%	2.6h	421.80	100%	4.6h
29	$2^{134}3^75^3$	$2^{19}3^3$	52.4MB	154	100%	5s	156.10	100%	20m	**125.70**	**100%**	**43m**
30	$2^{73}3^34^3$	$2^{31}3^4$	8.5MB	100	100%	2s	93.70	100%	2m	**73.80**	**100%**	**14m**
apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	92.5MB	173	100%	9s	191.60	100%	36m	**168.20**	**100%**	**1.7h**
bugzilla	$2^{49}3^14^2$	2^43^1	2.3MB	68	100%	1s	72.20	100%	22s	**49.50**	**100%**	**9m**
gcc	$2^{189}3^{10}$	$2^{37}3^3$	127.6MB	108	100%	10s	121	100%	43m	**81.80**	**100%**	**1.4h**
spins	$2^{13}4^5$	2^{13}	156.2kB	**98**	**100%**	**1s**	112.80	100%	2s	105.60	100%	3m
spinv	$2^{42}3^24^{11}$	$2^{47}3^2$	4.3MB	286	100%	2s	251.70	100%	2m	**238.90**	**100%**	**1.2h**
Segall et al. [27]												
Banking1	3^44^1	5^{112}	3.8kB	58	100%	2s	55.10	100%	0s	**45**	**100%**	**30s**
Banking2	$2^{14}4^1$	2^3	51.2kB	39	100%	1s	44.70	100%	0s	**30**	**100%**	**3m**
CommProtocol	$2^{10}7^1$	$2^{10}3^{10}4^{12}5^{24}6^{30}7^{30}8^{12}$	26.0kB	49	100%	3s	50.30	100%	0s	**41**	**100%**	**3m**
Concurrency	2^5	$2^43^15^2$	0.9kB	**8**	**100%**	**1s**	**8**	**100%**	**0s**	**8**	**100%**	**0s**
Healthcare1	$2^63^25^16^1$	2^33^{18}	31.9kB	105	100%	1s	107.50	100%	0s	**96**	**100%**	**9s**
Healthcare2	$2^53^64^1$	$2^13^65^{18}$	48.2kB	67	100%	1s	68.40	100%	0s	**54.80**	**100%**	**3m**
Healthcare3	$2^{16}3^64^55^16^1$	2^{31}	918.8kB	209	100%	1s	205.70	100%	15s	**177.10**	**100%**	**41m**
Healthcare4	$2^{13}3^{12}4^65^26^17^1$	2^{22}	2.2MB	294	100%	1s	309	100%	39s	**274.90**	**100%**	**53m**
Insurance	$2^63^15^16^211^113^117^131^1$	-	1.3MB	6866	100%	1s	6861.10	100%	3m	**6858.40**	**100%**	**15m**
NetworkMgmt	$2^24^15^310^211^1$	2^{20}	189.4kB	1125	100%	1s	1107.70	100%	4s	**1100.40**	**100%**	**2m**
ProcessorComm1	$2^33^64^6$	2^{13}	172.7kB	163	100%	1s	144.10	100%	2s	**131.60**	**100%**	**3m**
ProcessorComm2	$2^33^{12}4^85^2$	1^42^{121}	1015.3kB	161	100%	2s	169.30	100%	11s	**145.50**	**100%**	**31m**
Services	$2^33^45^28^210^2$	$3^{386}4^2$	365.6kB	963	100%	6s	**926.80**	**100%**	**13s**	926.80	100%	5.7h
Storage1	$2^13^14^15^1$	4^{95}	1.8kB	**25**	**100%**	**2s**	**25**	**100%**	**0s**	**25**	**100%**	**0s**
Storage2	3^46^1	-	5.1kB	74	100%	0s	71.50	100%	0s	**54**	**100%**	**1s**
Storage3	$2^93^15^36^18^1$	$2^{38}3^{10}$	184.4kB	239	100%	1s	239.20	100%	3s	**222**	**100%**	**9m**
Storage4	$2^53^74^15^26^27^110^113^1$	2^{24}	1.0MB	990	100%	1s	970.40	100%	28s	**916.40**	**100%**	**15m**
Storage5	$2^53^85^36^28^19^110^211^1$	2^{151}	2.1MB	**1879**	**100%**	**4s**	1936.10	100%	3m	1000.50	96%	12h
SystemMgmt	$2^53^45^1$	$2^{13}3^4$	26.7kB	60	100%	1s	58.10	100%	0s	**45**	**100%**	**2s**
Telecom	$2^53^14^25^16^1$	$2^{11}3^14^9$	43.2kB	126	100%	1s	125.20	100%	0s	**120**	**100%**	**5s**
Yu et al. [31]												
RL-A-mod	$2^53^44^75^46^57^48^112^3$	$1^{12}2^{491}3^{345}$	8.6MB	1132	100%	16s	1079.40	100%	4m	**1069.20**	**100%**	**7.8h**
RL-B-mod	$2^83^24^35^36^19^110^112^214^320^124^137^1$	$1^82^{1127}3^{277}4^{1755}5^{1064}6^{2048}$	16.4MB	14977	100%	4m	**13319.40**	**100%**	**3.1h**	4954	92%	12h
Yamada et al. [29]												
Company2	$2^63^48^4$	$1^22^{35}3^{89}4^{54}5^{34}6^{20}7^{34}8^{16}9^4$	247.9kB	**424**	**100%**	**15s**	432.50	100%	7s	427.20	100%	54m

Table 2 Test suite size, Percentage of tuple coverage and Time for $t = 4$ and $t = 5$. In bold the method with better results with the lexicographic criteria (coverage percentage, number of tests, exhausted time). For the coverage percentage enough precision was taken into account. Resources: 12GB memory and 12h timeout.

		$t=4$													$t=5$											
		mem	ACTS			BOT-its			PBOT-its			PRBOT-its			mem	ACTS			BOT-its			PBOT-its			PRBOT-its		
family	inst		tests	%	time	tests	%	time	tests	%	time	tests	%	time		tests	%	time	tests	%	time	tests	%	time	tests	%	time
Cohen et al.	1	1.4GB	**1680**	**100%**	**9m**	-	-	-	1721.30	100%	2.6h	1111.70	47%	12h	74.2GB	-	-	-	-	-	-	**1603.10**	**15%**	**12h**	938	3%	12h
	2	1.0GB	873	100%	4m	-	-	-	846.60	100%	1.5h	**736.70**	**100%**	**6.9h**	48.8GB	**3852**	**100%**	**7.2h**	-	-	-	1144.90	35%	12h	954.50	11%	12h
	3	7.5MB	212	100%	2s	253.90	100%	2m	253.90	100%	2m	**176.80**	**100%**	**1.1h**	99.3MB	593	100%	7s	747.60	100%	58m	749.60	100%	46m	**547.20**	**100%**	**6.2h**
	4	147.8MB	374	100%	10s	433.90	100%	1.1h	430.40	100%	55m	**339**	**100%**	**3.8h**	4.2GB	**1323**	**100%**	**13m**	-	-	-	1610.70	100%	3.9h	1025.50	49%	12h
	5	14.1GB	**2442**	**100%**	**2.5h**	-	-	-	2514.70	100%	11.3h	1111.10	41%	12h	1.3TB	-	-	-	-	-	-	**1603.90**	**1%**	**12h**	1546.70	1%	12h
	6	422.4MB	491	100%	47s	-	-	-	557	100%	43m	**426.70**	**100%**	**1.7h**	16.0GB	**1644**	**100%**	**1.0h**	-	-	-	833.50	60%	12h	811.60	38%	12h
	7	7.1MB	93	100%	2s	112.50	100%	54s	112.50	100%	55s	**88.90**	**100%**	**27m**	94.2MB	244	100%	9s	300.20	100%	20m	295.80	100%	17m	**235.20**	**100%**	**2.0h**
	8	2.9GB	1988	100%	25m	-	-	-	**1826**	**100%**	**5.3h**	1089.80	47%	12h	184.2GB	-	-	-	-	-	-	**1993.80**	**5%**	**12h**	1448.90	1%	12h
	9	172.9MB	268	100%	15s	347.10	100%	54m	345.80	100%	50m	**228.60**	**100%**	**2.9h**	5.2GB	**829**	**100%**	**11m**	-	-	-	1140.70	100%	5.8h	760.90	67%	12h
	10	7.2GB	**2063**	**100%**	**1.2h**	-	-	-	2128.50	99%	12h	1285.30	46%	12h	579.4GB	-	-	-	-	-	-	**1589.90**	**2%**	**12h**	1231.30	1%	12h
	11	1.4GB	**1885**	**100%**	**10m**	-	-	-	1984.10	100%	2.4h	1776.60	47%	12h	74.0GB	-	-	-	-	-	-	**3752.40**	**9%**	**12h**	3409.40	3%	12h
	12	6.1GB	**1465**	**100%**	**47m**	-	-	-	1491.80	100%	6.1h	1393.70	81%	12h	475.5GB	-	-	-	-	-	-	**1270.80**	**4%**	**12h**	922.40	1%	12h
	13	4.0GB	1040	100%	22m	-	-	-	1087.90	100%	4.6h	**958.40**	**100%**	**8.2h**	273.4GB	-	-	-	-	-	-	**1508.90**	**5%**	**12h**	1432.90	2%	12h
	14	1.1GB	1163	100%	5m	-	-	-	1250.20	100%	1.3h	**1145.10**	**100%**	**10.4h**	52.1GB	**5287**	**100%**	**9.0h**	-	-	-	2127.90	27%	12h	1998.10	8%	12h
	15	171.5MB	770	100%	20s	819.90	100%	2.5h	815.40	100%	2.2h	**710.10**	**100%**	**10.1h**	5.1GB	**2992**	**100%**	**28m**	-	-	-	3335.10	100%	6.5h	1563	45%	12h
	16	691.4MB	453	100%	2m	-	-	-	539.50	100%	1.1h	**411.10**	**100%**	**3.5h**	29.7GB	**1614**	**100%**	**2.1h**	-	-	-	655.50	35%	12h	624.10	21%	12h
	17	4.5GB	1514	100%	32m	-	-	-	1446.70	100%	3.9h	**1331.80**	**100%**	**8.6h**	325.2GB	-	-	-	-	-	-	**1861**	**4%**	**12h**	1643.70	2%	12h
	18	6.0GB	2145	100%	52m	-	-	-	**2020.90**	**100%**	**4.8h**	1052	42%	12h	465.5GB	-	-	-	-	-	-	**1391.70**	**3%**	**12h**	1346.80	1%	12h
	19	23.7GB	**2535**	**100%**	**5.1h**	-	-	-	689.10	99%	12h	665.70	38%	12h	2.5TB	-	-	-	-	-	-	**1415.90**	**1%**	**12h**	847.40	1%	12h
	20	11.1GB	**3278**	**100%**	**2.3h**	-	-	-	1424.20	99%	12h	1014	28%	12h	998.5GB	-	-	-	-	-	-	**2038.50**	**1%**	**12h**	1002	1%	12h
	21	796.2MB	1070	100%	3m	-	-	-	1149.90	100%	1.6h	**1039.20**	**100%**	**9.2h**	35.3GB	**4543**	**100%**	**4.8h**	-	-	-	3012.90	17%	12h	1079.70	7%	12h
	22	511.3MB	664	100%	1m	-	-	-	762.90	100%	59m	**611.30**	**100%**	**5.5h**	20.3GB	**2509**	**100%**	**2.0h**	-	-	-	939.30	75%	12h	766.30	26%	12h
	23	5.9MB	140	100%	2s	162.50	100%	55s	162.50	100%	57s	**122.50**	**100%**	**36m**	73.5MB	375	100%	6s	455.10	100%	21m	452.40	100%	17m	**347.60**	**100%**	**3.1h**
	24	2.9GB	2105	100%	25m	-	-	-	**2052.30**	**100%**	**4.2h**	1607.20	45%	12h	184.0GB	-	-	-	-	-	-	**2236.20**	**6%**	**12h**	1851.40	2%	12h
	25	5.3GB	**2673**	**100%**	**55m**	-	-	-	2681.90	100%	6.3h	1214.60	42%	12h	394.1GB	-	-	-	-	-	-	**1613**	**3%**	**12h**	1474.70	1%	12h
	26	1.1GB	1111	100%	5m	-	-	-	1054.50	100%	1.9h	**1004.50**	**100%**	**7.9h**	55.1GB	**5117**	**100%**	**9.3h**	-	-	-	1669.50	20%	12h	1167.80	3%	12h
	27	224.5MB	**1004**	**100%**	**35s**	1033.90	100%	4.5h	1034.20	100%	3.9h	931.80	60%	12h	7.2GB	**4379**	**100%**	**52m**	-	-	-	4486.80	99%	12h	1343.90	16%	12h
	28	22.7GB	**2888**	**100%**	**5.1h**	-	-	-	867.30	69%	12h	866.20	26%	12h	2.4TB	-	-	-	-	-	-	**1842.20**	**1%**	**12h**	1562.50	1%	12h
	29	5.1GB	681	100%	22m	-	-	-	758.70	100%	3.0h	**604.40**	**100%**	**7.7h**	374.4GB	-	-	-	-	-	-	**746.70**	**6%**	**12h**	712.90	2%	12h
	30	456.4MB	386	100%	56s	-	-	-	392.20	100%	40m	**313.90**	**100%**	**2.4h**	17.6GB	**1362**	**100%**	**1.1h**	-	-	-	517.40	64%	12h	497	37%	12h
	apache	10.9GB	838	100%	1.1h	-	-	-	944.30	100%	3.4h	**814.50**	**100%**	**9.3h**	971.3GB	-	-	-	-	-	-	**1027.80**	**3%**	**12h**	982.20	1%	12h
	bugzilla	79.4MB	242	100%	5s	275.70	100%	23m	277.30	100%	21m	**192**	**100%**	**1.7h**	1.9GB	752	100%	4m	-	-	-	932.60	100%	1.3h	**643.10**	**100%**	**6.8h**
	gcc	16.7GB	444	100%	1.2h	-	-	-	533.90	100%	5.6h	**364.80**	**100%**	**8.0h**	1.6TB	-	-	-	-	-	-	**436.40**	**3%**	**12h**	353.80	1%	12h
	spins	1.9MB	**393**	**100%**	**1s**	431.20	100%	36s	431.20	100%	37s	396.10	100%	2.3h	16.6MB	1449	100%	2s	1449.60	100%	14m	1448.80	100%	14m	**1360.30**	**100%**	**6.2h**
	spinv	181.2MB	1631	100%	38s	**1377.60**	**100%**	**5.9h**	1380.80	100%	4.6h	800.40	65%	12h	5.5GB	**8202**	**100%**	**1.1h**	-	-	-	1962.70	89%	12h	763	7%	12h
Segall et al.	Banking1	8.0kB	**139**	**100%**	**2s**	150.20	100%	0s	150.20	100%	0s	139	100%	3m	6.3kB	**212**	**100%**	**2s**	**212**	**100%**	**0s**	**212**	**100%**	**0s**	**212**	**100%**	**0s**
	Banking2	432.2kB	96	100%	1s	109.20	100%	3s	109.20	100%	3s	**87.30**	**100%**	**10m**	2.4MB	232	100%	1s	262.90	100%	29s	262.90	100%	30s	**225.30**	**100%**	**1.5h**
	CommProtocol	157.5kB	97	100%	3s	100.10	100%	1s	100.10	100%	1s	**86**	**100%**	**3m**	616.9kB	167	100%	4s	167	100%	5s	167	100%	5s	**162.50**	**100%**	**19m**
	Concurrency	1.2kB	**8**	**100%**	**1s**	**8**	**100%**	**1s**	**8**	**100%**	**1s**	**8**	**100%**	**1s**	0.6kB	**8**	**100%**	**1s**	**8**	**100%**	**1s**	**8**	**100%**	**1s**	**8**	**100%**	**1s**
	Healthcare1	201.2kB	341	100%	1s	331.60	100%	3s	331.60	100%	3s	**300**	**100%**	**21m**	801.8kB	814	100%	1s	829.40	100%	21s	829.40	100%	22s	**773.20**	**100%**	**6.6h**
	Healthcare2	379.6kB	220	100%	1s	233.50	100%	4s	233.50	100%	4s	**216.90**	**100%**	**52m**	1.9MB	708	100%	1s	716.80	100%	45s	716.80	100%	46s	**696.20**	**100%**	**6.0h**
	Healthcare3	21.4MB	1004	100%	4s	997.80	100%	15m	992.60	100%	15m	**904.70**	**100%**	**9.3h**	366.0MB	**4239**	**100%**	**58s**	-	-	-	4206.30	99%	12h	1846.20	57%	12h
	Healthcare4	73.2MB	**1644**	**100%**	**8s**	1792.50	100%	1.2h	1786.60	100%	1.0h	1035.20	77%	12h	1.7GB	**8204**	**100%**	**7m**	-	-	-	2843.30	23%	12h	2367	11%	12h
	Insurance	27.3MB	**75764**	**100%**	**4s**	69969.70	99%	12.0h	69377.70	99%	12.0h	67183.30	94%	12h	363.7MB	**494748**	**100%**	**53s**	-	-	-	27183.50	11%	12h	26578.20	6%	12h
	NetworkMgmt	2.0MB	6267	100%	2s	**6143.10**	**100%**	**3m**	6143.10	100%	3m	5276.20	99%	12h	12.6MB	**29272**	**100%**	**6s**	29764.60	100%	1.7h	29764.60	100%	1.7h	17096.20	93%	12h
	ProcessorComm1	2.1MB	670	100%	1s	638.30	100%	41s	638.30	100%	41s	**586.60**	**100%**	**5.1h**	18.5MB	**2588**	**100%**	**3s**	2589.20	100%	19m	2589.80	100%	19m	1088.10	95%	12h
	ProcessorComm2	24.3MB	744	100%	6s	789.20	100%	7m	791.40	100%	8m	**688.30**	**100%**	**7.1h**	424.0MB	**3094**	**100%**	**40s**	-	-	-	3333	100%	4.2h	1132	84%	12h
	Services	5.4MB	6855	100%	9s	6606.60	100%	16m	6606.60	100%	17m	**6539.20**	**100%**	**4.3h**	52.5MB	**37393**	**100%**	**22s**	17868.60	93%	12h	34818.20	99%	12h	5805.60	65%	12h
	Storage1	1.9kB	**25**	**100%**	**2s**	**25**	**100%**	**0s**	**25**	**100%**	**0s**	**25**	**100%**	**0s**	0.1kB	-	-	-	-	-	-	-	-	-	-	-	-
	Storage2	11.4kB	195	100%	0s	195.50	100%	0s	195.50	100%	0s	**162**	**100%**	**4s**	9.5kB	**486**	**100%**	**0s**	**486**	**100%**	**0s**	**486**	**100%**	**0s**	**486**	**100%**	**0s**
	Storage3	2.2MB	752	100%	2s	755.30	100%	54s	755.30	100%	55s	**686.90**	**100%**	**5.5h**	18.8MB	**2106**	**100%**	**4s**	2265.10	100%	14m	2269.40	100%	16m	938	97%	12h
	Storage4	23.2MB	**6636**	**100%**	**5s**	6729	100%	49m	6735.60	100%	50m	2407.60	94%	12h	372.2MB	**39490**	**100%**	**37s**	-	-	-	14598.40	97%	12h	5946	10%	12h
	Storage5	61.7MB	13292	100%	20s	**13183.10**	**100%**	**7.9h**	13183.60	100%	7.3h	1608	73%	12h	1.3GB	**78464**	**100%**	**9m**	-	-	-	5395.30	7%	12h	1790.90	2%	12h
	SystemMgmt	162.5kB	152	100%	1s	151.20	100%	1s	151.20	100%	1s	**135**	**100%**	**18s**	629.8kB	317	100%	1s	333.60	100%	6s	333.60	100%	6s	**285.70**	**100%**	**1.5h**
	Telecom	302.9kB	392	100%	1s	404	100%	3s	404	100%	3s	**361.40**	**100%**	**3m**	1.3MB	1110	100%	1s	1126.70	100%	31s	1126.70	100%	32s	**1044.80**	**100%**	**10.7h**
Yu et al.	RL-A-mod	438.0MB	**7912**	**100%**	**2m**	6680.40	99%	12h	6035.80	99%	12h	4228.10	93%	12h	15.6GB	**45775**	**100%**	**1.3h**	-	-	-	3531.80	2%	12h	3555.90	2%	12h
	RL-B-mod	956.6MB	**222846**	**100%**	**1.1h**	-	-	-	9091.50	11%	12h	8791.60	11%	12h	37.9GB	-	-	-	-	-	-	**5269.20**	**1%**	**12.0h**	5386.60	1%	11.9h
Yamada et al.	Company2	3.2MB	1398	100%	22s	**1393.50**	**100%**	**3m**	1393.50	100%	3m	1130.40	99%	12h	28.4MB	3412	100%	33s	**3360.60**	**100%**	**49m**	3364.20	100%	55m	1098.50	89%	12h

required by other auxiliary data structures or by the SAT solver called within BOT-its. Tables 1 and 2 show the result of our analysis under column *mem*. For $t = 4$ there are 20 out of the 58 instances that would consume more than 1GB. For $t = 5$ the memory consumption is greatly increased, as 23 of the 58 instances would consume more than 32GB (some of these instances would need more than 1TB). Therefore, it is obvious we can not aim to run any approach that explicitly considers all allowed t-tuples or tests at once under low memory requirements.

The third question we address is whether the Pool-based versions of BOT-its and RBOT-its are efficient compared to ACTS for $t = 4$ and $t = 5$. For both PRBOT-its and PBOT-its (as PRBOT-its but refine is deactivated) we consider a pool budget of 1GB (1278264 tuples for $t = 4$ and 721600 for $t = 5$). For $t = 4$ the combination of PBOT-its and PRBOT-its report better sizes than ACTS and BOT-its in 35 of the 58 instances. Finally, for $t = 5$ we found that ACTS and BOT-its can only report test suites for 39 and 18 instances respectively, while PBOT-its and PRBOT-its can report test suites for all the 57 instances[8].

Overall, we found that ACTS reports MCACs in 49 more instances than RBOT-its and PRBOT-its. However, we may be observing an horizon effect, as RBOT-its and PRBOT-its with the given resources are able to improve the results of ACTS in 89 out of 107 instances where both these algorithms and ACTS reach 100% of coverage, where ACTS only obtains better results in 8 (the remaining 10 are ties).

Regarding run times, ACTS is significantly faster than BOT-its, RBOT-its, PBOT-its and PRBOT-its. However, ACTS will report the same suboptimal solution with more available run time. In contrast, RBOT-its, and PRBOT-its can get better solutions if we increase the timeout for the MaxSAT call related to the refining process.

A more fine grained analysis on the new methods reveals the following insights.

We observe PBOT-its subsumes BOT-its, as it can obtain an MCAC on the same instances as BOT-its plus 23 and 7 more for $t = 4$ and $t = 5$ respectively. Regarding MCAC sizes we observe similarities with the results reported by BOT-its. Regarding run times we found that PBOT-its can obtain MCACs slightly faster than BOT-its.

Finally, we also note that with enough run time, RBOT-its and PRBOT-its algorithms would subsume BOT-its and PBOT-its respectively. In particular, results show that the *refine* approach can reduce the sizes on 92 out of the 106 instances where all these algorithms are able to obtain an MCAC, while for the remaining 14 instances they report the same sizes. In these particular cases, we observe that *refine* has not been able to improve the size of the window within the given time constraints, so these results could be improved by tuning the time limits, the MaxSAT solver's parameters or even using a different MaxSAT solver.

To conclude this section, it seems we can confirm the goodness of the PRBOT-its algorithm. We have shown how the *refine* method can be used to improve the sizes of the reported suboptimal MCACs. Additionally, we extended the practical usage of algorithm BOT-its to strengths higher than $t = 3$.

8 Conclusions and Future Work

Bugs or failures involving 4 or 5 parameters (even more) do exist and are likely to arise in complex systems. We have provided an effective approach to compute MCACs of such strength with low memory requirements. This low memory consumption plus the partitioning nature of the Pool based approach opens the avenue for more practical parallelized approaches.

[8] For instance *Storage1* it is not possible to report an MCAC for $t = 5$ as it only has 4 parameters.

References

1 Carlos Ansótegui and Felip Manyà. 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*, pages 1–15, 2004.

2 Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete maxsat approaches for combinatorial testing. *arXiv*, abs/2105.12552, 2021. `arXiv:2105.12552`.

3 Carlos Ansótegui, Jesus Ojeda, António Pacheco, Josep Pon, Josep M. Salvia, and Eduard Torres. Optilog: A framework for sat-based systems. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2021. `doi:10.1007/978-3-030-80223-3_1`.

4 Carlos Ansótegui, Idelfonso Izquierdo, Felip Manyà, and José Torres Jiménez. A max-sat-based approach to constructing optimal covering arrays. *Frontiers in Artificial Intelligence and Applications*, 256:51–59, 2013.

5 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. MaxSAT Evaluation 2019 : Solver and Benchmark Descriptions. Technical Report Department of Computer Science Report Series B-2019-2, University of Helsinki, 2019.

6 Mutsunori Banbara, Haruki Matsunaka, Naoyuki Tamura, and Katsumi Inoue. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

7 Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. In *SAT Competition 2013*, 2013.

8 Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.

9 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

10 Mehra N. Borazjany, Linbin Yu, Yu Lei, Raghu Kacker, and Rick Kuhn. 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*, pages 591–600, 2012.

11 Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 146–155, 2005.

12 Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 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.

13 Jacek Czerwonka. Pairwise testing in real world. In *Proc. of the Twenty-fourth Annual Pacific Northwest Software Quality Conference, 10-11 October 2006, Portland, Oregon*, pages 419–430, 2006.

14 Feng Duan, Yu Lei, Linbin Yu, Raghu N. Kacker, and D. Richard Kuhn. Optimizing ipog's vertical growth with constraints based on hypergraph coloring. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2017, Tokyo, Japan, March 13-17, 2017*, pages 181–188, 2017.

15 Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, January 2006. Publisher: IOS Press.

16 Ian P Gent and Peter Nightingale. A new encoding of alldifferent into sat. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

17 Arnaud Gotlieb, Aymeric Hervieu, and Benoit Baudry. Minimum pairwise coverage using constraint programming techniques. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 773–774, 2012. doi:10.1109/ICST.2012.174.

18 Aymeric Hervieu, Dusica Marijan, Arnaud Gotlieb, and Benoit Baudry. Practical minimization of pairwise-covering test configurations using constraint programming. *Information and Software Technology*, 71:129–146, 2016. doi:10.1016/j.infsof.2015.11.007.

19 Brahim Hnich, Steven Prestwich, and Evgeny Selensky. Constraint-based approaches to the covering test problem. In Boi V. Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints*, pages 172–186, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

20 Brahim Hnich, Steven D. Prestwich, Evgeny Selensky, and Barbara M. Smith. Constraint Models for the Covering Test Problem. *Constraints*, 11(2):199–219, July 2006.

21 Serdar Kadioglu. Column generation for interaction coverage in combinatorial software testing, 2017. arXiv:1712.07081.

22 D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004.

23 Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. Publisher: IOS Press.

24 Elizabeth Maltais and Lucia Moura. Finding the best CAFE is np-hard. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 356–371, 2010.

25 Toru Nanba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E95.A(9):1501–1505, 2012.

26 Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):1–29, 2011.

27 Itai Segall, Rachel Tzoref-Brill, and Eitan Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, page 254–264, New York, NY, USA, 2011. Association for Computing Machinery.

28 Evgeny Selensky. CSPLib problem 045: The covering array problem. http://www.csplib.org/Problems/prob045.

29 Akihisa Yamada, Armin Biere, Cyrille Artho, Takashi Kitamura, and Eun-Hye Choi. Greedy combinatorial test case generation using unsatisfiable cores. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 614–624, New York, NY, USA, 2016. Association for Computing Machinery.

30 Akihisa Yamada, Takashi Kitamura, Cyrille Artho, Eun-Hye Choi, Yutaka Oiwa, and Armin Biere. 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*, pages 1–10, 2015.

31 L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn. Constraint handling in combinatorial test generation using forbidden tuples. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–9, 2015.