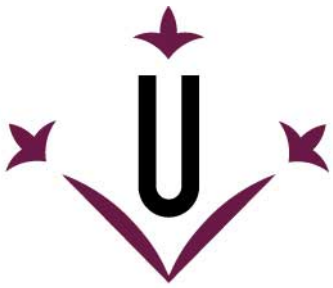
SAT-Based Combinatorial Testing

Eduard Torres Montiel



Universitat de Lleida

LOG

Logic & Optimization Group

TESI DOCTORAL

SAT-Based Combinatorial Testing

Eduard Torres Montiel

Memòria presentada per optar al grau de Doctor per la Universitat de Lleida
Programa de Doctorat en Enginyeria i Tecnologies de la Informació

Director i Tutor

Dr. Carlos Ansótegui Gil

2022

Acknowledgements

First of all, I would like to thank my advisor, Carlos Ansótegui, for his unconditional support throughout my thesis. Thank you for all the countless hours you spent with me discussing ideas, for those *deadline* nights finishing those conference papers, and for being more than an advisor, a leader and a friend.

I would also like to thank my friends Josep Pon, who thought of me when Carlos asked him about *"a student to collaborate in the research group"*, and Gerard Rovira, for all his support while we were undergraduate and master students, and also to my colleagues Josep Alòs and Josep M. Salvia in the Logic & Optimization Group. Thank you for all the good moments that we spent together.

Thanks also to two of the most important persons in my life, my sister Marta and my life partner Aina. They understood me every time I said *"I can't go, I need to finish these experiments"*, and for comprehending how important was for me finishing this thesis.

Last but not least, I would like to thank my whole family, as without them this thesis would have not been possible. To the best of my knowledge, I will be the first person in my family that receive a Doctoral degree. Fortunately, I could have a good Primary and High School education in the public system, something that none of my grandparents could have. Later, I pursued my Computer Engineering degree at the University of Lleida, something that none of my parents could afford. Finally, I spent four of the best years of my life doing research thanks to the FPU grant that we obtained in our research group.

For me, this thesis is much more than a personal achievement. It is a familiar achievement that would not have been possible without the hard work of my grandparents Ángeles, Paco, Dolores and Manuel, as well as my parents Luci and Eduardo, who made sacrifices for my sister and me, so we could aim for a brighter future. It is also an achievement of our society, as it would have been impossible for me to pursue higher education without the public education and grants systems. To this end, I know I will work and fight for a future where people like me could have (at least) the same opportunities I had.

"Que morir és no viure lluitant." Zoo - Vull

Abstract

In this thesis, we address the Combinatorial Testing (CT) problem through the application of Satisfiability technology. The goal of CT is to provide a test suite that is used to capture most of the errors or bugs in a System Under Test (SUT). In particular, we present several algorithms for the generation of Mixed Covering Arrays with Constraints (MCAC). First, we show how to apply Maximum Satisfiability technology to generate optimal and suboptimal MCACs. Second, we show how to effectively compute high-strength MCACs (considering strengths $t > 4$) by applying new incomplete algorithms and parallel solutions. Third, we design and implement a new benchmark generator for Combinatorial Testing that uses instances from the Satisfiability problem to generate new crafted SUTs. We also provide insights and recipes on which MCAC algorithm should be used for a particular SUT. All the algorithms designed during this thesis have been implemented into the *CTLog* package. Finally, we describe our contributions to the OptiLog Python framework for the development of SAT-based applications that we used to create *CTLog*.

Resumen

En esta tesis abarcamos el problema de Combinatorial Testing (CT) mediante la aplicación de la tecnología de Satisfactibilidad. El objetivo de CT es proveer un conjunto de pruebas que permita capturar la mayoría de los errores en un Sistema Bajo Prueba (SUT). En particular, presentamos varios algoritmos para generar Covering Arrays Mixtos y con Restricciones (MCACs). En primer lugar, mostramos como aplicar la tecnología de la Máxima Satisfactibilidad para generar MCACs óptimos y subóptimos. En segundo lugar, mostramos como computar MCACs con *strengths* altas de manera efectiva (considerando *strengths* $t > 4$) mediante la aplicación de nuevos algoritmos incompletos y soluciones paralelas. En tercer lugar, presentamos el diseño y la implementación de un nuevo generador de instancias para Combinatorial Testing que utiliza instancias del problema de la Satisfactibilidad para generar SUTs artificiales. Adicionalmente, hemos sido capaces de extraer distintas *recetas* sobre qué algoritmo de MCAC aplicar a un SUT determinado. Todos los algoritmos diseñados durante esta tesis han sido implementados en el paquete *CTLog*. Finalmente, describimos nuestras contribuciones en el *framework* de Python *OptiLog*, pensado para el desarrollo de aplicaciones basadas en la tecnología SAT, el cual ha sido utilizado para crear *CTLog*.

Resum

En aquesta tesi ens centrem en el problema de Combinatorial Testing mitjançant l'aplicació de la tecnologia de Satisfactibilitat. L'objectiu de CT és proveir d'un conjunt de proves que permeti capturar la majoria dels errors en un Sistema Sota Prova (SUT). En particular, presentem diversos algorismes per a generar Covering Arrays Mixtes amb Restriccions (MCACs). En primer lloc, mostrem com aplicar la tecnologia de la Màxima Satisfactibilitat per a generar MCACs òptims i subòptims. En segon lloc, mostrem com computar MCACs amb *strengths* altes de manera efectiva (considerant *strengths* $t > 4$) mitjançant l'aplicació de nous algorismes incomplets i solucions paral·leles. En tercer lloc, presentem el disseny i la implementació d'un nou generador d'instàncies per a Combinatorial Testing que utilitza instàncies del problema de la Satisfactibilitat per a generar SUTs artificials. Addicionalment, hem sigut capaços d'extraure diverses *receptes* sobre quin algorisme de MCAC aplicar a un SUT determinat. Tots els algorismes dissenyats durant aquesta tesi s'han implementat dins del paquet *CTLog*. Finalment, descrivim les nostres contribucions al *framework* de Python *OptiLog*, pensat per al desenvolupament d'aplicacions basades en la tecnologia SAT, el qual s'ha utilitzat per tal de crear *CTLog*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Think for a moment about all the pieces of complex technology that unstoppably are surrounding us. From our computer, our car, or any industrial process, technology plays an essential role in our society. In an ideal world, all this technology would be error-free and fully reliable, especially critical ones. This is a challenging task, currently underestimated, underregulated and underfunded.

From the software engineering perspective, we can find examples where bugs in the software have been responsible of disastrous consequences. In the Therac-25 radiation therapy incident, a bug in the code produced the death of at least five patients in the 1980s after administering excessive doses of beta radiation[1]. In the Northeast blackout incident in 2003, a race condition in the software of an electrical network control room produced a power outage that affected more than 50 million people around Ontario and the United States[2]. Another more recent bug that affected some users of Valve's Steam Linux client in 2015 was the accidental deletion of all the user's files and folders, which was caused by a common unsafe mistake in shell script programming[3].

To try to catch as many of these failures before they are produced, testing has become an essential part of the development process of a system. Although it is usually unfeasible to exhaustively test a system, there exist techniques to systematically test a system using a reasonable amount of resources. In particular, in this thesis, we focus on detecting errors that are produced by the interaction of the *parameters* of a system, where a *parameter* is defined as a setting on a system that can be set to different values.

The Combinatorial Testing (CT) problem [90] encompasses testing techniques in which multiple combinations of the input parameters are used to perform testing of the system, where a system can be a program, a circuit, a package that integrates several pieces of software, a GUI interface, a cloud application, etc. This problem requires exploring the parameter space of the system by iteratively testing different settings of the parameters to detect errors, bugs or faults.

Exploring all the parameter space exhaustively is, in general, out of reach. In particular, if a system has a set of parameters P, the number of different full assignments is $\prod_{p \in P} g_p = \mathcal{O}\left(g^{|P|}\right)$, where g_p is the cardinality of the domain of parameter p and g is the cardinality of the greatest domain.

The good news is that, in practice, there is no need to explore all the parameter space to detect errors, bugs or faults. We just need to *cover* a portion of the possible

[1] http://sunnyday.mit.edu/papers/therac.pdf

[2] https://www.energy.gov/oe/services/electricity-policy-coordination-and-implementation/august-2003-blackout

[3] https://www.energy.gov/oe/services/electricity-policy-coordination-and-implementation/august-2003-blackout

parameter combinations. For example, most software errors (75%-80%) are caused by certain individual parameters or by the interaction of just two of them [70].

To cover that portion of parameter combinations exhaustively, Covering Arrays (CAs) play an important role in CT. Given a set of parameters P and a strength t, a Covering Array $CA(N; t, P)$ is a test suite of N tests that guarantees to cover all the possible interactions of t parameters (referred as t-tuples). In general, since executing a test in the system has a cost, we will be interested in working with relatively small Covering Arrays.

Another relevant aspect of modern systems is the presence of *constraints* among their parameters that rule out some combinations of their values for which the system is intentionally not designed to work. We refer to these combinations as *not-allowed* or *forbidden*. In these cases, the system may not allow the user even to input this combination because of the user interface, or will warn the user the combination is not allowed or the system will just not react depending on how responsive it is. In any case, this is conceptually different from a combination that is *allowed* by the system, but unexpectedly causes a bug or an error that we can observe since it does not correspond with the expected behaviour.

For these cases, CT techniques must also ensure that none of the unsupported configurations represented by these constraints appear in any generated test. Therefore, the presence of constraints in a system provides another challenge for CT, as they must be properly handled by the CT algorithms.

Constraint programming approaches [93] are well-suited for handling constraints. Among them, SAT technology [33] provides a highly competitive generic problem approach for solving decision problems and MaxSAT technology [33] for optimization problems. In particular, the decision problem to be solved is translated into a SAT instance (a propositional formula) and a SAT solver is used to determine whether there is a solution, while the optimization problem is translated to MaxSAT and a MaxSAT solver is used to determine the optimum or a suboptimal solution. These techniques will play an essential role in the approaches we design and develop in this thesis.

As an example, consider the *GCC* compiler, which is an optimizing compiler produced by the GNU Project supporting various programming languages, hardware architectures and operating systems. *GCC* is one of the biggest free programs in existence, and the standard compiler for most projects related to GNU and the Linux kernel. Obviously, any bug in *GCC* can have a potential cascade effect on many other systems.

In particular, *GCC* has 189 input parameters of domain 2 and 10 input parameters of domain 3, as well as 40 user constraints that rule out forbidden combinations of the parameters. Checking all the parameter space would require $4.6 \cdot 10^{61}$ tests (settings to the input parameters) on the particular selected scenario (set of benchmarks where GCC is applied). Even if the test would only take one second when applied to the scenario, we would need the age of the universe to complete all the possible tests according to the parameter space of GCC.

However, we could reduce the number of required test scenarios by checking only the interactions of just 2 parameters. If we apply a naive enumeration approach, where at each test we only track one single interaction, we would need 82809 tests. However, by using CT techniques we can reduce this number to just 15 tests. If all these 15 tests do not produce any error we know that there is no interaction of two parameters of GCC that causes a failure or bug in our selected scenario.

In this thesis, we focus on designing and developing algorithms for the generation of Mixed Covering Arrays with Constraints (MCAC). 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.

First, we present a Satisfiability (SAT)-based approach for building Mixed Covering Arrays with Constraints of minimum length, referred to as the Covering Array Number problem. In particular, we show how to apply Maximum Satisfiability (MaxSAT) technology by describing efficient encodings for different classes of complete and incomplete MaxSAT solvers to compute optimal and suboptimal solutions, respectively. Similarly, we show how to solve through MaxSAT technology a closely related problem, the Tuple Number problem, which we extend to incorporate constraints. For this problem, we additionally provide a new MaxSAT-based incomplete algorithm. The extensive experimental evaluation we carry out on the available Mixed Covering Arrays with Constraints benchmarks and the comparison with state-of-the-art tools confirm the good performance of our approaches.

Second, 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 and its parallel version that allows reducing run-time consumption. Thanks to these new developments we provide a tool for Combinatorial Testing in practical environments. The experimental results confirm the goodness of the approach, opening avenues for new practical applications.

Third, while there is an active research community working on developing CT tools, paradoxically little attention has been paid to making available enough resources to test the CT tools themselves. In particular, the set of available benchmarks to asses their correctness, effectiveness and efficiency is rather limited. We introduce a new generator of CT benchmarks that essentially borrows the structure contained in the plethora of available Combinatorial Problems from other research communities to create meaningful benchmarks. We additionally perform an extensive evaluation of CT tools with these new benchmarks. Thanks to this study we provide some insights on under which circumstances a particular CT tool should be used.

Finally, all the developments from this thesis have been conducted on top of OptiLog, a Python framework designed and implemented by our research group, that eases the development of SAT-based applications. This has led to several contributions related to the features added to OptiLog in the last years: a full redesign of the solvers module to support the dynamic loading of incremental SAT solvers with support for external libraries, a module for modelling problems into Non-CNF format with support for Pseudo Boolean constraints, a module for evaluating and parsing the results of applications, a module for automatic configuration (tuning) of any Python function or external tool, and the support for constrained execution of blackbox programs and SAT-heritage integration. Thanks to these enhancements OptiLog can become a swiss knife for SAT-based applications in academic and industrial environments.

1.1 Objectives

This thesis aims to contribute to the advancement of algorithms for computing Mixed Covering Arrays with Constraints (MCAC). To reach the main goal, we focus on the

following objectives:

1. Use MaxSAT technology to generate MCACs.
2. Develop algorithms to effectively compute high-strength MCACs (considering strengths $t > 4$).
3. Design and implement a new benchmark generator for Combinatorial Testing.
4. Contribute to the development of the OptiLog Python framework, one of the key pieces in the development of all the algorithms presented in this thesis.

1.2 Structure of this thesis

The first chapter introduces this thesis. In Chapter 2 we provide several preliminary definitions and examples to better understand the contents of this thesis, as well as the definitions of some state-of-the-art CT algorithms. Chapter 3 corresponds to Objective 1 and discusses how to apply MaxSAT technology to generate MCACs. In Chapter 4, which corresponds to Objective 2, we focus on generating high-strength MCACs (considering strengths $t \geq 4$) and presents new sequential and parallel algorithms that can build these kinds of MCACs. Chapter 5 corresponds to Objective 3 and describes a new benchmark generator for Combinatorial Testing. Finally, in Chapter 6, we present the contributions that we performed to the OptiLog Python framework. We conclude this thesis in Chapter 7.

1.3 Thesis Outputs

In this section, we describe the different outputs produced in this thesis. In the first part, we detail the publications which we have contributed and in the second part the software tools that we developed.

1.3.1 Publications

Here we present the list of research articles that we produced during the development of this thesis.

- 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.
- Carlos Ansótegui, Jesus Ojeda, and Eduard Torres. Building high strength mixed covering arrays with constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021
- Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. Optilog V2: model, solve, tune and run. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing,*

SAT 2022, August 2-5, 2022, Haifa, Israel, volume 236 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

- Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete MaxSAT approaches for combinatorial testing. Journal of Heuristics, 28(4):377–431, August 2022.
- Carlos Ansótegui and Eduard Torres. Effectively Computing High Strength Mixed Covering Arrays with Constraints. Submitted to *Artificial Intelligence Review*.
- Carlos Ansótegui and Eduard Torres. A Benchmark Generator for Combinatorial Testing. Submitted to *IEEE Transactions on Software Engineering*.

1.3.2 Tools

We implemented the different algorithms explained throughout this thesis in the *CTLog* tool, available at `http://ulog.udl.cat/static/doc/ctlog/html/index.html`. This tool includes the implementation of five state-of-the-art MCAC algorithms, plus a generator of CT benchmarks. It has been developed in the Python[4] programming language, and some of its most critical parts in Nim[5]. We used OptiLog, which provided efficient access to SAT solvers and SAT encoders in Python, aside from other important utilities such as the *running* module to run and parse experiments (see Chapter 6).

Overall, this tool consists of 74 Python files with more than 7500 lines of code and 6 Nim files with almost 1000 lines of code. The following list shows the MCAC algorithms that are implemented in *CTLog*.

- **maxsat-mcac**: MCAC generation using MaxSAT technology. This implementation is used in Chapter 3.
- **CALOT**: Implementation of the incremental SAT-based MCAC algorithm described in [101]. Also used in Chapter 3.
- **maxsat-its**: Greedy MCAC algorithm that builds MCACs by solving the Tuple Number problem iteratively using MaxSAT technology. Also used in Chapter 3.
- **PRBOT-its**: Adaptation of the *Algorithm 5* in [100], which is a greedy algorithm to build MCACs. We extended this algorithm with its *pool* version (the *P* in *PRBOT-its*) and with the *refinement* version (the *R* in *PRBOT-its*). We also implement the *parallel* version of *PRBOT-its*. See Chapter 4 for an in-depth description and experimentation of these algorithms.
- **IPOG**: Implementation of the IPOG algorithm [73], another greedy algorithm for building MCACs. It is used in Chapter 5.

All these algorithms support the *ACTS*[6] and *Extended ACTS*[7] input formats, and outputs MCACs using an standardized `.csv` format. Additionally, we provide for

[4] `https://www.python.org/`

[5] `https://nim-lang.org/`

[6] The full specification of the ACTS format can be found in `https://csrc.nist.rip/groups/SNS/acts/documents/acts_user_guide_2.92.pdf`

[7] See Section 5.1.1

all these algorithms an option to *validate* the generated MCACs by using an efficient MCAC checker that we implemented in Nim.

Finally, we briefly describe the benchmark generator tool that is also included in *CTLog*.

- **SUT-G**: Generator of CT benchmarks with constraints by using an input SAT instance. The generator will try to find satisfiable *subproblems* on the input SAT instance that match the user's requirements in terms of constraints *hardness*. See Chapter 5 for a full description of the generator.

Aside from *CTLog*, we also contributed to the development of OptiLog, available at `http://ulog.udl.cat/static/doc/optilog/html/index.html`. Overall, OptiLog contains over 10000 lines of Python code and over 5000 lines of C/C++ code (excluding the code of the SAT solver engines). We present in Chapter 6 the main contributions that we performed in the second iteration of this project.

Chapter 2

State of the art

In this chapter, we formally introduce all the preliminary concepts that will be required in the following chapters. We will start introducing concepts related to the Satisfiability and Maximum Satisfiability problems. After that, we will present several Combinatorial Testing definitions and some of the most relevant state-of-the-art algorithms for building Mixed Covering Arrays With Constraints.

2.1 The Satisfiability and Maximum Satisfiability Problems

In this section, we introduce the definitions related to the Satisfiability and Maximum Satisfiability problems.

Definition 1. 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 2. 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.

Example 1. *The following Weighted Partial MaxSAT instance* $\phi = \{(x_1, 1), (x_2, 1), (x_1 \vee x_2, 2), (\neg x_1 \vee \neg x_2, \infty)\}$ *contains 3 soft clauses and 1 hard clause.*

Definition 3. 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 4. 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 5. 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.

Example 2. *Given* $I = \{x_1 = 0, x_2 = 0\}$ *and the instance* ϕ *in Example 1, the* $cost(I, \phi)$ *is 4.*

Definition 6. 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 $cost(I, \phi) = 0$.

Example 3. *The optimal cost of the instance ϕ in Example 1 is 1.*

Definition 7. 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 $cost(I, \varphi) = 0$.

Definition 8. An unsatisfiable core is a subset of clauses of a SAT instance that is unsatisfiable.

Definition 9. 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.

Example 4. *Given the SAT instance $\varphi = \{(x_1 \vee x_2), (x_1 \vee \neg x_2 \vee x_3 \vee x_4)\}$ and the partial truth assignment $I = \{x_1 = 0\}$, $UP(I, \varphi)$ simplifies φ to $\{(x_3 \vee x_4)\}$ and extends I to $\{x_1 = 0, x_2 = 1\}$.*

Definition 10. Let A and B be SAT instances.
$A \models B$ denotes that A entails B, i.e. all assignments satisfying A also satisfy B.
It holds that $A \models B$ iff $A \wedge \neg B$ is unsatisfiable.
$A \vdash_{UP} B$ denotes that, for every clause $c \in B$, $UP(A \wedge \neg c)$ derives a conflict.
If $A \vdash_{UP} B$ then $A \models B$.

Definition 11. We refer as SAT solver to 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.

Definition 12. A *pseudo-Boolean* (PB) constraint is a Boolean function of the form $\sum_{i=1}^{n} q_i l_i \diamond k$, where k and the q_i are integer constants, l_i are literals, and $\diamond \in \{<, \leq, =, \geq, >\}$.

Definition 13. A Cardinality (Card) constraint is a PB constraint where all q_i are equal to 1. An *At-Most-One* (AMO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \leq 1$. An *At-Least-One* (ALO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i \geq 1$. An *Exactly-One* (EO) constraint is a cardinality constraint of the form $\sum_{i=1}^{n} l_i = 1$.

Code fragment SATSolver shows the interface of a modern incremental SAT solver. The input instance is added to the solver with functions *add_clause* and *add_retractable* (in case the clause can be retracted) (lines 5 and 6), which operate on a single clause, while functions *add* and *retract* operate on a set of clauses. The last two functions are overloaded to ease the usage of SAT solvers within MaxSAT solvers (lines 17 and 20). Variable *n_vars* indicates the number of variables of the input formula (line 1).

Code SATSolver: Members and functions interface

```
   #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 add_retractable(c : clause)
       #Adds the clause c to the retractable list of clauses of the solver
7  function retract_clause(c : clause)
       #Retracts the clause c from the solver's list of retractable clauses
8  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
9      return status
10 function propagate(assumps : literals)
       #Performs Unit Propagation over the input formula given assumps
11     return The set of propagated literals
12 function set_seed(seed : int)
       #Sets the Random Number Generator seed of the SAT solver to the given seed
13 function add(φ : SAT formula)
14     foreach c_i ∈ φ do sat.add_clause(c_i)
15 function retract(φ : SAT formula)
16     foreach c_i ∈ φ do  sat.retract_clause(c_i)
   #Overloaded functions for SAT-based MaxSAT algorithms
17 function add(φ : Weighted Partial MaxSAT formula)
18     foreach (c_i, w_i) ∈ φ do
19         if w_i = ∞ then sat.add_clause(c_i) else sat.add_retractable(c_i)
20 function retract(φ : Weighted Partial MaxSAT formula)
21     foreach (c_i, w_i) ∈ φ do  if w_i ≠ ∞ then sat.retract_clause(c_i)
```

Function *solve* (line 8) returns UNSAT (SAT) if the input formula is unsatisfiable (satisfiable) and sets variable *core* (*model*) to the corresponding unsatisfiable core (model), as well as the attribute $n\_conflicts$ to the number of conflicts produced in this call. Additionally, this function supports the *assumps* argument, which is a list of literals and allows to place an *assumption* on the truth value of each literal before function *solve* is called. Regarding the *propagate* method (line 10), it will apply Unit Propagation over the input instance, and can also receive a list of assumptions *assumps* as in the *solve* function.

SAT solvers also tend to expose the *set_seed* function (line 12), which allows fixing the Random Number Generator seed in the SAT solver to a given *seed*. Finally, modern SAT solvers also support an incremental solving mode, which allows keeping the learnt clauses across calls to the function *solve*.

2.2 The Combinatorial Testing Problem

The Combinatorial Testing (CT) problem [90] addresses the question of how to efficiently verify the proper operation of a system. This can be achieved by exploring the parameter space of the system and iteratively testing different settings of the parameters to detect errors, bugs or faults.

In this section, we present all the required concepts related to Combinatorial Testing. We also include some examples to better understand these ideas.

Definition 14. 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.

Example 5. *Consider an online service that offers a web page and a mobile application that share the same code base. We want to find potential failures given different OS, platforms, screen resolutions and orientations. The definition of this example SUT S regarding its parameters S_P is the following:*

$$\begin{aligned} OS\ (OS) &\in \{Linux\ (L),\ Windows\ (W),\ Mac\ (M),\ iOS\ (i),\ Android(A)\} \\ Platform\ (Pl) &\in \{Firefox\ (F),\ Safari\ (S),\ Chrome\ (C),\ App\ (A)\} \\ Resolution\ (Re) &\in \{4K\ (K),\ FHD\ (F),\ HD\ (H),\ WXGA\ (W)\} \\ Orientation\ (Or) &\in \{Portrait\ (P),\ Landscape\ (L)\} \end{aligned}$$

Notice that in this scenario we also have a set of SUT constraints S_φ:

$$((OS = L) \vee (OS = W) \vee (OS = M)) \rightarrow ((Or = L) \wedge (Pl \neq A)) \tag{2.1}$$

$$(Pl = S) \rightarrow ((OS = M) \vee (OS = i)) \tag{2.2}$$

$$((OS = i) \vee (OS = A)) \rightarrow (Re \neq K) \tag{2.3}$$

Whenever the OS is Linux, Windows *or* Mac, *the Orientation must be* Landscape *and the Platform cannot be* App *(equation (2.1)). When the Platform is* Safari, *the OS must be* Mac *or* iOS *(equation (2.2)). Finally, for* iOS *and* Android *the Resolution cannot be* 4K *(equation (2.3)).*

Definition 15. 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.

Example 6. *A test case for the SUT model in Example 5 is $\{(OS, W), (Pl, F), (Re, K), (Or, L)\}$. $\{Os, Pl\}$ is a parameter tuple and $\{(OS, W), (Pl, F)\}$ is a t-tuple for $t = 2$.*

Definition 16. 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 $\mathcal{T}_a = \{\tau \mid \tau \not\models \neg S_\varphi\}$.

We refer to the set of allowed t-tuples as $\mathcal{T}_a^{t,S} = \{\tau \mid \tau \not\models \neg S_\varphi\}$, to the set of forbidden t-tuples as $\mathcal{T}_f^{t,S} = \{\tau \mid \tau \models \neg S_\varphi\}$, and to the whole set of t-tuples in the SUT model S as $\mathcal{T}^{t,S} = \mathcal{T}_a \cup \mathcal{T}_f$. When there is no ambiguity, we refer to $\mathcal{T}_a^{t,S}, \mathcal{T}_f^{t,S}, \mathcal{T}^{t,S}$ as $\mathcal{T}_a, \mathcal{T}_f, \mathcal{T}$, respectively.

Example 7. *The t-tuple $\{(OS, i), (Re, K)\}$ is a forbidden tuple. There are 82 t-tuples ($|\mathcal{T}|$) for $t = 2$ in the SUT of Example 5. 69 of these 82 t-tuples are allowed ($|\mathcal{T}_a|$) and 13 are forbidden ($|\mathcal{T}_f|$).*

Definition 17. 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 Y *covers* a value tuple τ (i.e., $\tau \subseteq$ Y) if there exist a test case $v \in$ Y s.t. $v \models \tau$. We refer to $v \not\models \tau$ ($\tau \not\subseteq$ Y) when a test case (test suite) *does not cover* τ.

Example 8. *The test case $v = \{(OS, W), (Pl, F), (Re, K), (Or, L)\}$ covers the following t-tuples for $t = 2$:*

$$\{(OS, W), (Pl, F)\}, \{(OS, W), (Re, K)\}, \{(OS, W), (Or, L)\},$$
$$\{(Pl, F), (Re, K)\}, \{(Pl, F), (Or, L)\}, \{(Re, K), (Or, L)\}$$

Definition 18. 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.

Example 9. *Table 2.1 shows an MCAC for the SUT S in Example 5*

Definition 19. The Covering Array Number, $CAN(t, S)$, is the minimum N for which there exists an MCAC $CA(N; t, S)$. An upper bound $ub^{CAN(t,S)}$ for $CAN(t, S)$ is an integer such that $ub^{CAN(t,S)} \geq CAN(t, S)$, and a lower bound $lb^{CAN(t,S)}$ is an integer such that $CAN(t, S) > lb^{CAN(t,S)}$.

When there is no ambiguity, we refer to $ub^{CAN(t,S)}$ ($lb^{CAN(t,S)}$) as ub (lb).

Example 10. *Table 2.2 shows an optimal $CA(N; t, S)$ for the SUT model in Example 5. In this case $CAN(2, S) = 21$.*

Definition 20. The Tuple Number, $T(N; t, S)$, is the maximum number of t-tuples that can be covered by a set of N tests for a SUT model S. An upper bound $ub^{T(N;t,S)}$ for $T(N; t, S)$ is an integer such that $ub^{T(N;t,S)} \geq T(N; t, S)$, and a lower bound $lb^{T(N;t,S)}$ is an integer such that $T(N; t, S) > lb^{T(N;t,S)}$.

When there is no ambiguity, we refer to $ub^{T(N;t,S)}$ ($lb^{T(N;t,S)}$) as ub (lb).

Definition 21. The MCAC problem is to find an MCAC of size N.
The Covering Array Number problem is to find an MCAC of size $CAN(t, S)$.
The Tuple Number problem is to find a test suite of size N that covers $T(N; t, S)$ t-tuples.

The MCAC problem is a decision problem. The Covering Array Number and the Tuple Number problems, to which we refer in short as the $CAN(t, S)$ and $T(N; t, S)$ problems, respectively, are optimization problems.

2.3 Mixed Covering Arrays with Constraints Algorithms

In this section, we provide a brief review of some of the existing algorithms for building Mixed Covering Arrays with Constraints.

test	OS	Pl	Re	Or
v_1	L	F	F	L
v_2	L	C	H	L
v_3	W	F	W	L
v_4	W	C	K	L
v_5	M	F	H	L
v_6	M	S	W	L
v_7	M	C	F	L
v_8	i	F	H	P
v_9	i	S	F	P
v_{10}	i	C	W	P
v_{11}	i	A	F	L
v_{12}	A	F	H	P
v_{13}	A	C	W	L
v_{14}	A	A	F	P
v_{15}	L	F	K	L
v_{16}	M	S	K	L
v_{17}	i	S	H	L
v_{18}	A	A	H	L
v_{19}	i	A	W	L
v_{20}	W	F	F	L
v_{21}	W	F	H	L
v_{22}	L	F	W	L

TABLE 2.1: $CA(22; 2, S)$ for the SUT in example 5.

test	OS	Pl	Re	Or
v_1	L	C	K	L
v_2	L	F	F	L
v_3	L	C	H	L
v_4	L	C	W	L
v_5	W	F	K	L
v_6	W	C	F	L
v_7	W	F	H	L
v_8	W	F	W	L
v_9	M	S	K	L
v_{10}	M	S	F	L
v_{11}	M	C	H	L
v_{12}	M	F	W	L
v_{13}	i	C	F	P
v_{14}	i	S	H	P
v_{15}	i	S	W	P
v_{16}	A	A	F	L
v_{17}	A	A	H	P
v_{18}	A	F	W	P
v_{19}	i	A	W	L
v_{20}	A	C	H	L
v_{21}	i	F	H	L

TABLE 2.2: $CA(21; 2, S)$ for the SUT in example 5. This corresponds to the $CAN(2, S)$.

2.3.1 The In-Parameter Order General Algorithm (IPOG)

The In-Parameter Order General Algorithm (IPOG) [73] is one of the most widely-used algorithms for building suboptimal MCACs. It can be found implemented in different frameworks such as ACTS [34].

The original algorithm did not allow the presence of SUT constraints. However, there exist several variations where SUT constraints are supported [103, 104]. In this section, we describe the version in [104], which uses a CP solver to handle the SUT constraints.

Roughly speaking, the IPOG algorithm, iteratively builds an MCAC of strength t on a subset of the params in S_P. Given an order of the parameters S_P, at the kth step, the algorithm has already an MCAC for the first $k-1$ params and extends it to an MCAC on the first k params.

Algorithm IPOG shows the pseudocode. We start by sorting the S_p parameters (line 1). Then, we initialize the working test suite Y with the trivial MCAC of strength t on the first t parameters, i.e., a test for each allowed t-tuple.

The main loop (line 3) iterates on the sorted list of parameters P starting at the $t+1$th parameter. In line 4 we compute all the t-tuples that can be composed with the current parameter p and any $t-1$-tuple in Y and, store them in ρ. These are the new tuples that need to be allocated to the test suite. To do that we *extend* horizontally the current test suite and if needed, *fill the empties* and *grow* vertically the test suite by adding new tests.

From lines 5 to 10 we perform the horizontal extension. We just traverse every test in Y and select for the current test v which is the value for the current parameter p, that in combination with any set of $t-1$ values already in the test, covers more t-tuples in ρ and update v and ρ accordingly.

Of course, we double-check the current partially constructed test is consistent with S_φ. This is performed by checking whether the partial assignment represented

by the values in the test under construction can be extended to a satisfying assignment in S_φ.

It may be the case that, for a given test, there is more than one possible value for p but all the new t-tuples that could be obtained with these values have been already covered in a previous test. Then, we leave empty the value for the parameter p at this test, and *fill this empty* opportunistically later.

If we have already covered all the new tuples we can proceed to the next parameter (line 10).

It may be the case that by extending horizontally all the tests in Y we may have not covered all the new tuples in ρ. Then, we have two ways to cover the rest. We traverse the remaining t-tuples in ρ and check whether we can allocate the given t-tuple τ taking advantage of the *empties* (lines 13-18). Finally, if τ can not be allocated in the existing test suite Y, we just add a new test and τ to this test and leave the rest of the parameters in the test with an empty value (line 20).

Algorithm IPOG: IPOG Algorithm

Input : SUT model S, strength t
Output: Test suite Y

1 $P \leftarrow argsort_{p \in S_p} d(p)$
2 $Y \leftarrow CAN(t, \langle \{P_1, ..., P_t\}, S_\varphi \rangle)$
3 **for** p **in** P_{t+1} ... $P_{|P|}$ **do**
  `# Initialize tuples pool`
4 $\rho \leftarrow \{\tau \mid \tau \in d(p)$ combined with all the tuples of size $t-1$ in Y$\}$
  `# Horizontal extension`
5 **for** v **in** Y **do**
6 Choose best v $\in d(p)$ s.t. $v \cup \{(p,\text{v})\}$ covers more tuples in ρ and is consistent with S_φ
7 $v \leftarrow v \cup \{(p,\text{v})\}$
8 $\rho_v \leftarrow \{\tau \mid (\tau \in \rho) \wedge (v \models \tau)\}$
9 $\rho \leftarrow \rho \setminus \rho_v$
10 **if** $|\rho| = \emptyset$ **then break**
11 **if** $|\rho| \neq \emptyset$ **then**
12 **for** $\tau \mid (\tau \in \rho) \wedge (\tau$ *consistent with* $S_\varphi)$ **do**
  `# Fill the empties`
13 covered $\leftarrow$ false
14 **for** v **in** Y **do**
15 **if** $\tau \cup v$ *is possible and consistent with* S_φ **then**
16 $v \leftarrow v \cup \tau$
17 covered $\leftarrow$ true
18 **break**
  `# Vertical growth`
19 **if** *covered = false* **then**
20 $Y \leftarrow Y \cup \{v \; s.t. \; v \models \tau\}$
21 **return** Y

2.3.2 *Algorithm 5* and *BOT-its* Algorithms

There is another approach for building MCACs. Instead of building an MCAC by adding one parameter at a time such as in IPOG (see Section 2.3.1) we can build an MCAC One Test at a Time (also referred to as OTAT [35]).

Several algorithms that implement the OTAT approach, such as PICT [41]. However, we find other algorithms also based on OTAT that focus on the efficient handling of the SUT constraints.

In [100] the authors describe several of these algorithms, being *Algorithm 5* the most relevant. This algorithm has been reworked in our recent work [7] through the *BOT-its* algorithm and its variations.

Essentially, these algorithms build one test at a time, ensuring that at the end this test case is consistent with the SUT constraint, and *amending* the parts that are not consistent. Additionally, the *forbidden* tuples are lazily detected and removed by using unsatisfiable cores, which greatly reduces the employed resources for this task.

Chapter 4 describes in detail these algorithms and proposes several variations to improve the sizes of the generated MCACs and to scale this approach to larger instances and strengths.

Chapter 3

Mixed Covering Arrays with Constraints through Maximum Satisfiability

As we introduced in Chapter 1, we are focusing on building Mixed Covering Arrays with Constraints (MCACs). More formally, a Covering Array $CA(N; t, P)$ (where t is the strength and P the set of parameters) is a test suite of N tests that guarantee to cover all the possible interactions of t parameters (referred as t-tuples). Since executing a test in the system has a cost, we are interested in working with relatively small covering arrays. We refer to the minimum N for which a $CA(N; t, P)$ exists as the Covering Array Number, denoted by $CAN(t, P)$. In particular, we are interested in building an optimal CA, i.e., a covering array of length $CAN(t, P)$. Notice that it is guaranteed that the number of tests required to cover all t-way parameter combinations, for fixed t, grows logarithmically in the number of parameters [40], which indicates that optimal or near-optimal covering arrays can be used in practical terms. The computational challenge is to build optimal CAs in a reasonable time frame.

There exist several greedy approaches that tackle the problem of building minimum MCACs, such as PICT [41], based on the OTAT framework [35], and ACTS [34], based on the IPOG algorithm [44]. One downside of these approaches is that they become more inefficient as the hardness of the set of forbidden interactions increases. Therefore, we are more interested in constraint programming approaches, which are better suited for handling constraints. For example, CALOT [101] is a tool for building MCACs based on Satisfiability (SAT) technology [32] that can handle constraints efficiently.

Within constraint programming techniques [93], SAT technology provides a highly competitive generic problem approach for solving decision problems. In particular, the decision problem to be solved is translated into a SAT instance (a propositional formula) and a SAT solver is used to determine whether there is a solution. In this chapter, we will *review* in detail the CALOT tool, which essentially solves a sequence of SAT instances to compute an optimal MCAC. Each SAT instance in the sequence encodes the decision query of whether there exists an MCAC of a certain length. By iteratively bounding the length, the optimum can be determined.

Since the problem of computing minimum MCACs is, in essence, an optimization problem, we also consider its reformulation into the Maximum Satisfiability (MaxSAT) problem [32], which is an optimization version of the SAT problem.

We show empirically that MaxSAT approaches outperform ACTS and CALOT (the state-of-the-art) once the suitable MaxSAT encodings are used. We evaluate both complete or exact MaxSAT solvers (certify optimality) and incomplete MaxSAT

solvers (provide suboptimal solutions). In particular, we show that while complete MaxSAT solvers perform similar to CALOT (substantially in contrast to previously reported experiments with MaxSAT solvers [101]), incomplete MaxSAT solvers obtain better suboptimal solutions and faster than ACTS and CALOT on many instances. This confirms the practical interest of incomplete MaxSAT approaches because, in real environments, we are mainly concerned with obtaining the best possible solution within a given budget of runtime.

Having confirmed the good performance of MaxSAT approaches for computing minimum MCACs, we explore another related problem, the Tuple Number (TN) Problem. Informally, the TN problem is to determine the minimum set of missing t-tuples in a test suite of N tests, or the maximum set of t-tuples that these N tests cover. This problem is related to the Optimal Shortening Covering Arrays (OSCAR) problem [36] (which is NP-hard), where given a matrix of tests the goal is to find a submatrix of a fixed number of tests and parameters that maximizes the number of covered t-tuples. These *shortened* covering arrays have been used to improve the initialization of metaheuristic approaches for Covering Arrays (without SUT constraints).

In this chapter, we explore (for the first time) the *Mixed* and *with Constraints* variants of the TN problem, assessing the performance of complete and incomplete MaxSAT approaches. Obviously, this problem is of interest when $N < CAN(t, P)$[1]. We additionally present another incomplete approach based on MaxSAT technology to which we refer as MaxSAT Incremental Test Suite (Maxsat ITS), that *incrementally* builds the test suite with the help of a MaxSAT query that aims to maximize the coverage of allowed tuples at every step.

The Covering Array Number problem is concerned with reporting solutions with the least number of tests. From a practical point of view, whether we are satisfied with suboptimal solutions will depend on the cost of the tests. This cost basically includes the cost of generating the tests (computational resources) and the cost of testing the system. In particular, when the cost is too prohibitive in terms of our budget, and we are satisfied with covering a statistically significant portion of the tuples, we aim to solve (even suboptimally) the Tuple Number problem. Therefore, there exist real-world scenarios where all the approaches described in this chapter are of practical interest.

The rest of the chapter is structured as follows: Section 3.1 defines different SAT encodings and sections 3.2 and 3.3 describe techniques to make the SAT encodings more efficient. Section 3.4 introduces the incremental SAT algorithm CALOT for computing minimum MCACs. Subsequently, Section 3.5 defines MaxSAT encodings and Section 3.6 describes how to efficiently apply MaxSAT solvers. For the Tuple Number problem, Section 3.7 defines a MaxSAT encoding and Section 3.8 presents a new incomplete approach using MaxSAT solvers. To assess the impact of the presented approaches, Section 3.9 reports on an extensive experimental investigation on the available MCAC benchmarks.

3.1 The MCAC problem as SAT

In this section, we present the SAT encoding described in [101] to decide whether there exists a $CA(N; t, S)$ for a given SUT model $S = \langle P, \varphi \rangle$. It is similar to previous encodings described in [64, 65, 25, 87, 14].

[1] For $N \geq CAN(t, P)$, the Tuple Number problem essentially corresponds to determine the number of allowed tuples in the corresponding MCAC problem.

In the following, we list the set of constraints that define the SAT encoding and describe the semantics of the propositional variables they refer to. To encode each constraint, we assume that AMO and EO cardinality constraints are translated into CNF through the regular encoding [6, 56] and the typical transformations [97] of $\rightarrow$ and $\leftrightarrow$ are implicitly applied [2].

First, we define variables $x_{i,p,v}$ to be true iff test case i assigns value v to parameter p, and state that each parameter in each test case takes exactly one value as follows (where $[N] = \{1, \ldots, N\}$):

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

Second, as described in [88], to enforce the SUT constraints φ, for each test case i, we add the CNF formula that encodes the constraints of φ into SAT and substitute each appearance of the pair (p, v) in φ by the corresponding literal on the propositional variable $x_{i,p,v}$ for each test case i.

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

Third, we introduce propositional variables c^i_τ and state that if they are true, then tuple τ must be covered at test i, by forcing the variables p in the test case to be assigned to the value specified in τ, as follows:

$$\bigwedge_{i \in [N]} \bigwedge_{\tau \in \mathcal{T}_a} \bigwedge_{(p,v) \in \tau} (c^i_\tau \rightarrow x_{i,p,v}) \tag{CX}$$

Notice that only t-tuples that can be covered by a test case are encoded, i.e., $\tau \in \mathcal{T}_a$. In section 3.2, we discuss how to detect the t-tuples forbidden by the SUT constraints.

Finally, we state that every t-tuple $\tau \in \mathcal{T}_a$, must be covered at least by one test case, as follows:

$$\bigwedge_{\tau \in \mathcal{T}_a} \bigvee_{i \in [N]} c^i_\tau \tag{C}$$

Proposition 1. Let $Sat^{N,t,S}_{CX}$ be $X \wedge C \wedge CX \wedge SUTX$. $Sat^{N,t,S}_{CX}$ is satisfiable iff a $CA(N; t, S)$ exists.

Inspired by the incremental SAT approach in [101] (see section 3.4), we present another encoding where C and CX are replaced by CCX:

$$\bigwedge_{i \in [N]} \bigwedge_{\tau \in \mathcal{T}_a} \bigwedge_{(p,v) \in \tau} (c^i_\tau \rightarrow c^{i-1}_\tau \vee x_{i,p,v}) \quad \text{(a) } (CCX)$$

$$\bigwedge_{\tau \in \mathcal{T}_a} c^N_\tau \quad \text{(b) } (CCX)$$

$$\bigwedge_{\tau \in \mathcal{T}_a} (c^N_\tau \rightarrow \neg c^0_\tau) \quad \text{(c) } (CCX)$$

[2] We replace $A \rightarrow B$ with $(\neg A \vee B)$, and $A \leftrightarrow B$ with $(A \rightarrow B) \wedge (A \leftarrow B)$ where A and B are disjunctions of literals.

Variables c_τ^i have now a different semantics, i.e., if they are true, τ is covered by test case i or by any lower test case j, where $1 \leq j \leq i$ (equation a). In order to guarantee that τ will be covered by some test, notice that we just need to force c_τ^N to be true and c_τ^0 to be false (variables c_τ^0 are additionally included in the encoding). This can be achieved by adding the unit clauses c_τ^N (equation b) and the implication $c_\tau^N \rightarrow \neg c_\tau^0$ (equation c) for every allowed tuple τ.

The seasoned reader may wonder why we do not simply replace equation (c) by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$. Indeed, this is possible. First, notice that UP on the conjunction of equations (b) and (c) will derive exactly the same. Second, for encoding some problems where it is not mandatory to cover all the tuples (see section 3.7 on encoding the Tuple Number problem), we have to erase equation (b) from CCX and also guarantee that if a tuple τ is not covered in an optimal solution, i.e., c_τ^N has to be False, then the related clauses in CCX have to be satisfied (these are hard clauses) and, if possible, to be trivially satisfied, i.e., without requiring search. Equation (c) eases this case for all the scenarios in section 3.7. Notice that, once c_τ^N is False, clauses in equation (c) are trivially satisfied and, by setting the remaining c_τ^i vars to True, clauses in equation (a) are also trivially satisfied.

Proposition 2. Let $Sat_{CCX}^{N,t,S}$ be $X \wedge CCX \wedge SUTX$. $Sat_{CCX}^{N,t,S}$ is satisfiable iff a $CA(N;t,S)$ exists.

Remark 1. There are some variations of equation (a) in CCX that can be beneficial when using some SAT solvers, as we will see in section 3.9.1. For example, we can use full implication instead of half implication in equation (a), i.e., $(c_\tau^i \leftrightarrow c_\tau^{i-1} \vee x_{i,p,v})$, or we can even use $(c_\tau^i \rightarrow c_\tau^{i-1} \vee x_{i,p,v}) \wedge (c_\tau^i \leftarrow c_\tau^{i-1})$. Also, we can consider full implication in equation (c) and, for some of the problems analyzed in section 3.9.1, we can even replace equation (c) by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$.

Example 11. *We show how to build $Sat_{CX}^{N=22,t=2,S}$ for the SUT in Example 5, where $N = 22$ is an upper bound ub for this SUT (see section 3.2).*

To encode the X constraint, we add:

$$
\begin{aligned}
x_{1,OS,L} + x_{1,OS,W} + x_{1,OS,M} + x_{1,OS,i} + x_{1,OS,A} &= 1,\\
x_{1,Pl,F} + x_{1,Pl,S} + x_{1,Pl,C} + x_{1,Pl,A} &= 1,\\
x_{1,Re,K} + x_{1,Re,F} + x_{1,Re,H} + x_{1,Re,W} &= 1,\\
x_{1,Or,P} + x_{1,Or,L} &= 1,\\
&\vdots\\
x_{22,OS,L} + x_{22,OS,W} + x_{22,OS,M} + x_{22,OS,i} + x_{22,OS,A} &= 1,\\
x_{22,Pl,F} + x_{22,Pl,S} + x_{22,Pl,C} + x_{22,Pl,A} &= 1,\\
x_{22,Re,K} + x_{22,Re,F} + x_{22,Re,H} + x_{22,Re,W} &= 1,\\
x_{22,Or,P} + x_{22,Or,L} &= 1
\end{aligned}
\qquad \text{(Ex. X)}
$$

Next, for each test $(1, ..., 22)$, we encode the SUT constraints SUTX:

$$\begin{gathered}
(x_{1,OS,L} \vee x_{1,OS,W} \vee x_{1,OS,M}) \rightarrow (x_{1,Or,L} \wedge \neg x_{1,Pl,A}) \\
x_{1,Pl,S} \rightarrow (x_{1,OS,M} \vee x_{1,OS,i}) \\
(x_{1,OS,i} \vee x_{1,OS,A}) \rightarrow \neg x_{1,Re,K} \\
\vdots \\
(x_{22,OS,L} \vee x_{22,OS,W} \vee x_{22,OS,M}) \rightarrow (x_{22,Or,L} \wedge \neg x_{22,Pl,A}) \\
x_{22,Pl,S} \rightarrow (x_{22,OS,M} \vee x_{22,OS,i}) \\
(x_{22,OS,i} \vee x_{22,OS,A}) \rightarrow \neg x_{22,Re,K}
\end{gathered} \tag{Ex. SUTX}$$

Notice that these constraints are not in CNF. Their translation would be the following:

$$\begin{gathered}
(x_{i,OS,L} \vee x_{i,OS,W} \vee x_{i,OS,M}) \rightarrow (x_{i,Or,L} \wedge \neg x_{i,Pl,A}) \equiv \\
((\neg x_{i,OS,L} \vee x_{i,Or,L}) \wedge (\neg x_{i,OS,L} \vee \neg x_{i,Pl,A}) \wedge (\neg x_{i,OS,W} \vee x_{i,Or,L}) \wedge \\
(\neg x_{i,OS,W} \vee \neg x_{i,Pl,A}) \wedge (\neg x_{i,OS,M} \vee x_{i,Or,L}) \wedge (\neg x_{i,OS,M} \vee \neg x_{i,Pl,A}))
\end{gathered} \tag{CNF 2.1}$$

$$x_{i,Pl,S} \rightarrow (x_{i,OS,M} \vee x_{i,OS,i}) \equiv (\neg x_{i,Pl,S} \vee x_{i,OS,M} \vee x_{i,OS,i}) \tag{CNF 2.2}$$

$$\begin{gathered}
(x_{i,OS,i} \vee x_{i,OS,A}) \rightarrow \neg x_{i,Re,K} \equiv \\
((\neg x_{i,OS,i} \vee \neg x_{i,Re,K}) \wedge (\neg x_{i,OS,A} \vee \neg x_{i,Re,K}))
\end{gathered} \tag{CNF 2.3}$$

Finally, the encoding of the CX and C constraints is shown below. We identify the set of allowed tuples, ($\mathcal{T}_a$), as described in section 3.2. In particular, there are $|\mathcal{T}_a| = 69$ allowed tuples as we mention in Example 7.

$$\begin{array}{ccc}
c^1_{\tau_1} \rightarrow x_{1,OS,L}, & \ldots, & c^1_{\tau_{69}} \rightarrow x_{1,Re,W} \\
c^1_{\tau_1} \rightarrow x_{1,Pl,F}, & \ldots, & c^1_{\tau_{69}} \rightarrow x_{1,Or,L} \\
\vdots & \ddots & \vdots \\
c^1_{\tau_1} \rightarrow x_{22,OS,L}, & \ldots, & c^1_{\tau_{69}} \rightarrow x_{22,Re,W} \\
c^1_{\tau_1} \rightarrow x_{22,Pl,F}, & \ldots, & c^1_{\tau_{69}} \rightarrow x_{22,Or,L}
\end{array} \tag{Ex. CX}$$

$$(c^1_{\tau_1} \vee c^2_{\tau_1} \vee \cdots \vee c^{22}_{\tau_1}), \quad \ldots, \quad (c^1_{\tau_{69}} \vee c^2_{\tau_{69}} \vee \cdots \vee c^{22}_{\tau_{69}}) \tag{Ex. C}$$

To build $Sat^{N=22,t=2,S}_{CCX}$, we encode the CCX constraint instead of the C and CX constraints:

$$
\begin{array}{rcl}
c^1_{\tau_1} \rightarrow c^0_{\tau_1} \vee x_{1,OS,L}, & \ldots, & c^1_{\tau_{69}} \rightarrow c^0_{\tau_{69}} \vee x_{1,RE,W} \\
c^1_{\tau_1} \rightarrow c^0_{\tau_1} \vee x_{1,Pl,F}, & \ldots, & c^1_{\tau_{69}} \rightarrow c^0_{\tau_{69}} \vee x_{1,Or,L} \\
\vdots & \ddots & \vdots \qquad \text{(Ex. CCX a)} \\
c^{22}_{\tau_1} \rightarrow c^{21}_{\tau_1} \vee x_{22,OS,L}, & \ldots, & c^{22}_{\tau_{69}} \rightarrow c^{21}_{\tau_{69}} \vee x_{22,RE,W} \\
c^{22}_{\tau_1} \rightarrow c^{21}_{\tau_1} \vee x_{22,Pl,F}, & \ldots, & c^{22}_{\tau_{69}} \rightarrow c^{21}_{\tau_{69}} \vee x_{22,Or,L} \\
c^{22}_{\tau_1}, & \ldots, & c^{22}_{\tau_{69}} \qquad \text{(Ex. CCX b)} \\
c^{22}_{\tau_1} \rightarrow \neg c^0_{\tau_1}, & \ldots, & c^{22}_{\tau_{69}} \rightarrow \neg c^0_{\tau_{69}} \qquad \text{(Ex. CCX c)}
\end{array}
$$

Once we run a SAT solver on any of the previous SAT instances, if there exists a $CA(22;2,S)$*, it will return a satisfying truth assignment. To recover the particular* $CA(22;2,S)$ *implicitly found by the solver, we just need to check the assignment to the* $x_{i,p,v}$ *variables. For example, if* $x_{1,OS,L}$ *is True then parameter OS takes value Linux at test* 1*.*

3.2 Preprocessing for the MCAC problem

In the context of the Covering Array Number problem, we define an upper bound ub and a lower bound lb to be integers such that $ub \geq CAN(t,S) > lb$. When $ub = lb + 1$, we can stop the search and report ub as the minimum covering array number $CAN(t,S)$.

To get an initial value for ub, we can execute a greedy approach to obtain a suboptimal $CA(N;t,S)$ and set ub to N. For example, in the experiments, we use the tool ACTS [34] that supports Mixed Covering Arrays with Constraints. Moreover, a lower ub also implies a smaller initial encoding.

Additionally, by inspecting the solution, i.e., the test cases that certify the suboptimal $CA(N;t,S)$, we can compute which tuples are not covered, the set of *forbidden* tuples, since the suboptimal $CA(N;t,S)$ guarantees to cover all allowed t-tuples.

Furthermore, let r be the maximum number of allowed t-tuples associated with any parameter tuple of length t. Then, we can set $lb = r - 1$, since these r value tuples (mutually exclusive) need to be covered by different test cases.

Algorithm ForbiddenTuples: Detection of forbidden tuples.

```
1 Input: SUT model S, SAT solver sat
2 sat.add(Sat_CX^{N=1,t,S}[X, SUTX])
3 T_f = ∅
4 for τ ∈ T do
5 |   A ← {x_{1,p,v} | (p,v) ∈ τ}
6 |   if sat.solve(A) = UNSAT then T_f ← T_f ∪ {τ}
7 return T_f
```

To detect the forbidden tuples, we can apply the algorithm ForbiddenTuples. This algorithm tests, for every tuple τ (lines 4-6), if it is compatible with the SUT constraints (line 2) through a SAT query; if the solver results in unsatisfiability (line 6), the tuple is added to the set of forbidden tuples $\mathcal{T}_f$, which is ultimately returned by the algorithm (line 7).

For $t = 2$, which is already of practical importance [70], the experiments carried out in this chapter show that this detection process is negligible runtime-wise.

3.3 Symmetry Breaking for the MCAC problem

As [101], we fix the r t-tuples that conducted us to set the initial lb (see section 3.2) to test cases $\{1, \ldots, r\}$. This helps us break row symmetries for the first r test cases. We will refer to this as fixed-tuple symmetry breaking.

There are other alternatives. We can impose row symmetry breaking constraints as [47]; since each row (test) represents a number in base 2, we can add constraints to order the tests in monotonic increasing order, from test 0 to test $N - 1$. We can also apply, as explained above, fixed-tuple symmetry breaking to the first r tuples (first partition) and apply row symmetry breaking constraints to the remaining $ub - lb + 1$ test cases (second partition). Furthermore, we can impose an order among the tuples in the first partition and the second partition, so that if two sets share the same value for the fixed tuple, then the one representing the lower number must be in the first partition.

Our experimental analysis shows that fixed-tuple symmetry breaking is superior to any other of the mentioned alternatives. For lack of space, we restricted all the experiments to this symmetry breaking approach.

Example 12. *We show how to apply symmetry breaking to the SUT in Example 5.* (Os, Re) *is the parameter tuple with the largest number of allowed tuples, being* $|\mathcal{T}_a| = 18$*. To apply the fixed-tuple symmetry breaking variant, we just need to fix each allowed value tuple in a different test as shown below:*

$$\begin{aligned} & x_{1,Os,L} \wedge x_{1,Re,K} \\ & x_{2,Os,L} \wedge x_{2,Re,F} \\ & \vdots \\ & x_{18,Os,A} \wedge x_{18,Re,W} \end{aligned} \qquad \text{(Ex. SYM X)}$$

3.4 Solving the $CAN(t, S)$ problem with Incremental SAT

In this section, we present the CALOT algorithm, which is an incremental SAT approach for computing optimal covering arrays with SUT constraints described by [101]. The input to the algorithm is an upper bound ub (computed as in section 3.2), the strength t and the SUT model S. In line 2, the incremental SAT solver is initialized with the SAT instance $Sat_{CCX}^{N=ub,t,S}$. Additionally, breaking symmetries for the first $lb + 1$ tuples, as described in section 3.3, are added to the SAT solver. The output is the covering array number and an optimal model.

The algorithm works by iteratively decreasing the ub till it reaches $lb + 1$ (line 5) or the current SAT instance is unsatisfiable (line 6). To decrease the ub by one, the algorithm adds the set of unit clauses $\bigwedge_{\tau \in \mathcal{T}_a} c_\tau^{i-1}$ (line 7), which state that every t-tuple is covered by a test case with an index smaller than i.

There is a subtle detail in lines 9 and 10. Whenever the algorithm finds a new upper bound, variables $x_{i,p,v}$ related to the previous upper bound are fixed to the value in the last model found (b_{model} in line 8), so that these variables do not need to be decided in the next iterations. As [101] report, not fixing these variables can have

Algorithm CALOT: Algorithm 2 in [101]

Input: *upper bound ub, strength t, SUT model S*
$sat.add(Sat_{CCX}^{N=ub,t,S})$
Fix $lb+1$ value tuples to break symmetries (see Section 3.3)
$b_{model} \leftarrow \emptyset$
for $i = N, ..., lb+1$ **do**
  if $sat.solve() = UNSAT$ **then return** (i, b_{model})
  $sat.add(\bigwedge_{\tau \in \mathcal{T}} c_\tau^{i-1})$
  $b_{model} \leftarrow sat.model$
  for $\tau \in \mathcal{T}_a$ **do**
    for $(p, v) \in \tau$ **do** $b_{model}[x_{i,p,v}]$ **?** $sat.add(\{x_{i,p,v}\})$ **:** $sat.add(\{\neg x_{i,p,v}\})$
return $(lb+1, b_{model})$

some negative impact on the performance.

Remark 2. The original algorithm pseudocode is slightly different [101]. First, it assigns the i-th test at iteration i to the value it had in the previous model found instead of assigning the $i+1$-th test. This does not correspond to the description given in the text of the paper and may lead to an incomplete algorithm.

Second, the set of constraints (a) (CCX), described in [101], does not set c_τ^N to True as we do in this work, which makes the pseudocode perform a dummy first step that can cause to report a wrong optimum. We think that these are merely errors in the description, and we have fixed them. Since the tool CALOT is not available from the authors for reproducibility, we have tried to do our best to reproduce (or extend) the idea behind their work.

In section 3.6, we will see that this SAT incremental approach resembles how SAT-based MaxSAT algorithms behave [4, 85]. Actually, in contrast to [101], we show that MaxSAT technology can be effectively applied to solve Covering Arrays.

3.5 The $CAN(t, S)$ problem as Partial MaxSAT

[5] proposes an encoding into Partial MaxSAT to build covering arrays without constraints of minimum size. The main idea is to use an indicator variable u_i that is True iff test case i is used to build the covering array. 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:

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

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 (see section 3.2).

To avoid symmetries, it is also enforced that if test case $i+1$ belongs to the minimum covering array, so does the previous test case i:

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

Then, variables u_i are connected to variables c^i_τ, expressing that if we want test i to be the proof that τ is covered, then test i must be in the optimal solution [3]:

$$\bigwedge_{i\in[lb+2\ldots N]} \bigwedge_{\tau\in\mathcal{T}_a} (c^i_\tau \rightarrow u_i) \tag{CU}$$

Proposition 3. Let $PMSat^{N,t,S,lb}_{CX}$ be $SoftU \wedge BSU \wedge CU \wedge Sat^{N,t,S}_{CX}$. If $N \geq CAN(t,S)$, the optimal cost of the Partial MaxSAT instance $PMSat^{N,t,S,lb}_{CX}$ is $CAN(t,S) - (lb+1)$, otherwise it is ∞.

In order to build the Partial MaxSAT version of $Sat^{N,t,S}_{CCX}$, we just need to change how variables u_i are related to variables c^i_τ. This constraint reflects that if u_i is False (i.e., test i is not in the solution and, therefore, due to constraint BSU, none of the tests $> i$ cannot be in the solution either), then the tuple τ has to be covered at some test below i:

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

Proposition 4. Let $PMSat^{N,t,S,lb}_{CCX}$ be $SoftU \wedge BSU \wedge CCU \wedge Sat^{N,t,S}_{CCX}$. If $N \geq CAN(t,S)$, the optimal cost of the Partial MaxSAT instance $PMSat^{N,t,S,lb}_{CCX}$ is $CAN(t,S) - (lb+1)$, otherwise it is ∞.

Remark 3. In [5], variables u_i are instead connected to variables $x_{i,p,v}$ in the following way:

$$\bigwedge_{i\in[N]} \bigwedge_{p\in P} (u_i \leftrightarrow \bigvee_{v\in d(p)} x_{i,p,v}) \tag{XU}$$

This is a more compact encoding but it requires equation X to use an AMO constraint instead of an EO constraint.

Finally, we can convert these Partial MaxSAT instances into Weighted Partial MaxSAT modifying $SoftU$ as follows:

$$\bigwedge_{i\in[lb+2\ldots N]} (\neg u_i, w_i) \tag{WSoftU}$$

If we use $w_i = 2^{i-(lb+2)}$ we naturally introduce a lexicographical preference in the soft constraints. This is a key detail to alter the behaviour of SAT-based MaxSAT algorithms when solving Covering Arrays. If the MaxSAT solver applies the stratified approach [10] (see for more details section 3.6), it suffices to use $w_i = i - (lb+2) + 1$, i.e., to increase the weights linearly. This is of interest since a high number of tests in $WSoftU$ can result in too large weights for some MaxSAT solvers.

Proposition 5. Let $WPMSat^{N,t,S,lb}_{CCX}$ be $WSoftU \wedge BSU \wedge CCU \wedge Sat^{N,t,S}_{CCX}$.

If $N \geq CAN(t,S)$ and $w_i = 2^{i-(lb+2)}$ the optimal cost of the Weighted Partial MaxSAT instance $WPMSat^{N,t,S,lb}_{CCX}$ is $2^{CAN(t,S)-(lb+1)} - 1$, otherwise it is ∞.

If $N \geq CAN(t,S)$ and $w_i = i - (lb+2) + 1$ the optimal cost of the Weighted Partial MaxSAT instance $WPMSat^{N,t,S,lb}_{CCX}$ is $(1+n) \cdot n/2$ where $n = CAN(t,S) - (lb+1)$, otherwise it is ∞.

[3] Notice that τ could be covered by other tests but the respective c^i_τ variables be False.

Example 13. *We extend our working example to obtain the Partial MaxSAT and Weighted Partial MaxSAT encodings described in this section. We first describe how we encode SoftU (left) and BSU (right) constraints:*

$$\begin{array}{ll} (\neg u_{22}, 1) & \\ (\neg u_{21}, 1) & u_{22} \rightarrow u_{21} \\ (\neg u_{20}, 1) & u_{21} \rightarrow u_{20} \\ (\neg u_{19}, 1) & u_{20} \rightarrow u_{19} \end{array} \qquad \text{(Ex. SoftU and BSU)}$$

Recall that in our example $ub = 22$ and $lb = 17$ (see Examples 11 and 12). Therefore, we will have $N - (lb + 1) = 22 - (17 + 1) = 4$ u_i indicator variables.

To build the $PMSat_{CX}^{N=22,t=2,S,lb=17}$ instance we add to $Sat_{CX}^{N=22,t=2,S}$ the CU constraint:

$$\begin{array}{llll} c_{\tau_1}^{22} \rightarrow u_{22}, & \ldots, & c_{\tau_{69}}^{22} \rightarrow u_{22} \\ c_{\tau_1}^{21} \rightarrow u_{21}, & \ldots, & c_{\tau_{69}}^{21} \rightarrow u_{21} \\ c_{\tau_1}^{20} \rightarrow u_{20}, & \ldots, & c_{\tau_{69}}^{20} \rightarrow u_{20} \\ c_{\tau_1}^{19} \rightarrow u_{19}, & \ldots, & c_{\tau_{69}}^{19} \rightarrow u_{19} \end{array} \qquad \text{(Ex. CU)}$$

To build $PMSat_{CCX}^{N=22,t=2,S,lb=17}$ we add to $Sat_{CCX}^{N=22,t=2,S}$ the CCU constraint:

$$\begin{array}{llll} \neg c_{\tau_1}^{21} \rightarrow u_{22}, & \ldots, & \neg c_{\tau_{69}}^{21} \rightarrow u_{22} \\ \neg c_{\tau_1}^{20} \rightarrow u_{21}, & \ldots, & \neg c_{\tau_{69}}^{20} \rightarrow u_{21} \\ \neg c_{\tau_1}^{19} \rightarrow u_{20}, & \ldots, & \neg c_{\tau_{69}}^{19} \rightarrow u_{20} \\ \neg c_{\tau_1}^{18} \rightarrow u_{19}, & \ldots, & \neg c_{\tau_{69}}^{18} \rightarrow u_{19} \end{array} \qquad \text{(Ex. CCU)}$$

The weighted counterparts, $WPMSat_{CX}^{N=22,t=2,S,lb=17}$ and $WPMSat_{CCX}^{N=22,t=2,S,lb=17}$, need only to replace SoftU by WSoftU (using $w_i = i - (lb + 2) + 1$), as follows:

$$\begin{array}{l} (\neg u_{22}, 4) \\ (\neg u_{21}, 3) \\ (\neg u_{20}, 2) \\ (\neg u_{19}, 1) \end{array} \qquad \text{(Ex. WSoftU)}$$

To build the resulting MCAC from the MaxSAT solver truth assignment, we will discard the $x_{i,p,v}$ vars whose corresponding u_i is assigned to False (i.e. test i does not belong to the solution), and proceed as in Example 11.

3.6 Solving the $CAN(t, S)$ problem with MaxSAT

In this section, we show that SAT-based MaxSAT approaches can *simulate*[4] the CALOT algorithm, while the opposite is not true. This is an interesting insight since the

[4]By *simulate*, we informally refer to perform the same sequence of upper bound refinements and the same filtering based on unit propagation.

MaxSAT approach additionally provides the option of applying a plethora of MaxSAT algorithms.

Let us first introduce a short description of SAT-based MaxSAT algorithms. For further details, please consult [4, 85]. Roughly speaking, SAT-based MaxSAT algorithms proceed by reformulating the MaxSAT optimization problem into a sequence of SAT decision problems. Each SAT instance of the sequence encodes whether there exists an assignment to the MaxSAT instance with a cost less than or equal to a certain k. SAT instances with a k less than the optimal cost are unsatisfiable, while the others are satisfiable. The SAT solver is executed in incremental mode to keep the clauses learnt at each iteration over the sequence of SAT instances. Thus, SAT-based MaxSAT can also be viewed as a particular application of incremental SAT solving.

There are two main types of SAT-based MaxSAT solvers: (i) model-guided and (ii) core-guided. The first ones iteratively refine (decrease) the upper bound and guide the search with satisfying assignments (models) obtained from satisfiable SAT instances. The second ones iteratively refine (increase) the lower bound and guide the search with the unsatisfiable cores obtained from unsatisfiable SAT instances. Both have strengths and weaknesses, and hybrid approaches exist [13, 12].

3.6.1 The Linear MaxSAT Algorithm

The Linear algorithm [46, 72], described in Algorithm Linear, is a model-guided algorithm for WPMaxSAT. Let $\phi = \phi_s \cup \phi_h$ (line 1) be the input WPMaxSAT instance, where ϕ_s (ϕ_h) is the set of soft (hard) clauses in ϕ.

Algorithm Linear: Linear SAT-based algorithm

```
1  Input: Weighted Partial MaxSAT formula φ ≡ φ_s ∪ φ_h, SAT solver sat
2  sat.add(φ_h)
3  sat.add({c_i ∨ b_i | (c_i, w_i) ∈ φ_s})
4  ub ← Σ_{(c_i,w_i)∈φ_s} w_i + 1
5  pb ← Σ_{(c_i,w_i)∈φ_s} w_i · b_i ≤ ub − 1
6  sat.add(pb.to_cnf)
7  while True do
8      if sat.solve() = UNSAT then return (ub, sat.model)
9      ub ← Σ_{(c_i,w_i)∈φ_s} w_i · sat.model[b_i]
10     sat.add(pb.update(ub − 1))
```

At each iteration of the Linear algorithm, the SAT instance solved by the incremental SAT solver is composed of: (i) the hard clauses ϕ_h (line 2), which guarantee that any possible solution is a *feasible* solution; (ii) the reification of each soft clause $(c_i, w_i) \in \phi_s$ into clause $(c_i \vee b_i)$, where b_i is a fresh auxiliary variable which acts as a collector of the truth value of the soft clause (line 3); and (iii) the CNF translation of the PB constraint $\sum_{(c_i,w_i)\in\phi_s} w_i \cdot b_i \leq k$, where $k = ub - 1$ bounds the aggregated cost of the falsified soft clauses, i.e., the value of the objective function.

Initially, ub is set to $(\sum_{(c_i,w_i)\in\phi_s} w_i + 1)$ (line 4), that is semantically equivalent to ∞. Then, iteratively, if the incremental SAT solver returns satisfiable, ub is updated to $(\sum_{(c_i,w_i)\in\phi_s} w \cdot sat.model[b_i])$ (line 9) [5]; otherwise, ub is the optimal cost (line 8). If the input instance is unsatisfiable the algorithm returns $\sum_{(c_i,w_i)\in\phi_s} w_i + 1$ (i.e., ∞).

[5] $sat.model[b_i]$ is 1 if b_i is assigned to True in the model, otherwise it is 0.

A technical point to mention is that the PB constraint is translated into SAT thanks to an incremental PB encoding (line 5) so that whenever we tighten the upper bound, instead of retracting the original PB constraint and encode the new one, we just need to add some additional clauses (line 10). Additionally, if all the weights in the soft clauses are equal, instead of using an incremental PB encoding, we can use an incremental cardinality encoding for which more efficient encodings do exist.

Proposition 6. The Linear algorithm with Weighted Partial MaxSAT instance $WPMSat_{CCX}^{N,t,S,lb}$ as input can *simulate* the CALOT algorithm (excluding lines 9 and 10).

In the first place notice that in the worst case the Linear algorithm will decrease the current upper bound by one unit as the algorithm CALOT. Then, the key point establishing the connection of the Linear algorithm with the CALOT algorithm is to show that, given the same upper bound k to both algorithms, the Linear algorithm can propagate the same set of c_τ^{i-1} variables (line 7 in algorithm CALOT).

Let us recall that the Linear algorithm, with input $\phi \equiv WPMSat_{CCX}^{N,t,S,lb}$, will generate a sequence of SAT instances composed of the original hard clauses ϕ_h, the reification of the soft clauses $\bigwedge_{(c_i,w_i)\in\phi_s}(c_i \vee b_i)$, the translation to CNF of the PB constraint $\sum_{(c_i,w_i)\in\phi_s} w_i \cdot b_i \leq k$, where (c_i, w_i) represents the i-th soft clause in $WPMSat_{CCX}^{N,t,S,lb}$, i.e., $(\neg u_i, 2^{i-(lb+2)})$ when using the exponential increase, and the current upper bound k.

Proposition 7. If $\phi \equiv WPMSat_{CCX}^{N,t,S,lb}$, then
$CCU \wedge \bigwedge_{(\neg u_i, 2^{i-(lb+2)})\in\phi_s}(\neg u_i \vee b_i) \wedge \sum_{(\neg u_i, 2^{i-(lb+2)})\in\phi_s} 2^{i-(lb+2)} \cdot b_i \leq k \vdash_{UP} \bigwedge_{k<i\leq N+1}\bigwedge_{\tau\in\mathcal{T}_a} c_\tau^{i-1}$.

First of all, notice that the weight of a higher index test is strictly greater than the aggregated weights of the lower index tests. Given an upper bound k, an *efficient* CNF translation of the PB constraint will allow Unit Propagation (UP) to derive that all bs associated with soft clauses with a weight greater than k must be False. Then, from the set of clauses that reify the soft clauses (of the form $\neg u_i \vee b_i$), UP will also derive that the corresponding u_i vars must be False and, from the set of hard clauses CCU, UP will derive that the corresponding c_τ^{i-1} must be true.

If the input problem is a Partial MaxSAT instance, i.e., $PMSat_{CCX}^{N,t,S,lb}$ where the i-th soft clause is of the form $(\neg u_i, 1)$, the Linear algorithm uses a cardinality constraint instead of a PB constraint to bound the aggregated cost of the falsified soft clauses. In this case, we can only guarantee that $CCU \wedge \bigwedge_{(\neg u_i,1)\in\phi_s}(\neg u_i \vee b_i) \wedge \sum_{(\neg u_i,1)\in\phi_s} b_i \leq k \models \bigwedge_{k<i\leq N+1}\bigwedge_{\tau\in\mathcal{T}_a} c_\tau^{i-1}$. Notice that, given an upper bound k, UP cannot derive on $\sum_{(\neg u_i,1)\in\phi_s} b_i \leq k$ the set of b_is that must be False, because all correspond to soft clauses of equal weight.

CALOT algorithm cannot *simulate* the Linear Algorithm: While the CALOT algorithm decreases the upper bound by one at each iteration, the Linear algorithm can decrease it more aggressively. This is the case when it finds a model with a lower cost than $k-1$ (line 9), which can significantly reduce the number of calls to the SAT solver.

3.6.2 The WPM1 MaxSAT algorithm

The Fu&Malik algorithm [49] is a core-guided SAT-based MaxSAT algorithm for Partial MaxSAT instances. In contrast to the Linear algorithm, which uses the models to iteratively refine the upper bound, the Fu&Malik algorithm uses the unsatisfiable cores to refine the lower bound. In particular, the initial SAT instance φ_0 explored

by the Fu&Malik algorithm is composed of the hard clauses in the input MaxSAT instance ϕ_h plus the SAT clauses c_i *extracted* from the soft clauses (c_i, w_i). We refer to these c_i clauses as soft-indicator clauses.

At each iteration, if φ_k is satisfiable, the optimum is k. If φ_k is unsatisfiable, the clauses in the unsatisfiable core retrieved by the SAT solver are analyzed. If none of the clauses is a soft-indicator clause, the Partial MaxSAT formula is declared unsatisfiable and the algorithm stops. Otherwise, the core tells us that we need to relax the soft-indicator clauses, i.e., we need to violate more clauses. To construct the next instance, φ_{k+1}, each soft-indicator clause in the core of φ_k is relaxed with a fresh auxiliary variable b and a hard EO cardinality constraint is added on these new variables, indicating that at least one clause must be violated (this is what the core told us) and at most one clause is violated (this prevents jumping over the optimum).

The WPM1 algorithm [11, 83] is an extension of the Fu&Malik algorithm that solves Weighted Partial MaxSAT instances by applying the split rule for weighted clauses. In particular, we are interested in using the *Stratified* WPM1 algorithm (WPM1) [10], which clusters the input clauses according to their weights[6]. These clusters were originally named as strata in [10]. The algorithm incrementally merges the clusters solving the related subproblem until all clusters have been merged. In its simpler version, all the clauses in a cluster have the same weight (called the representative weight), and clusters are added in decreasing order with respect to the representative weight, but other strategies can also be applied [10].

Algorithm WPM1: Stratified WPM1

```
1  Input: Weighted Partial MaxSAT formula φ, SAT solver sat
2  φ_wk, φ_re, status ← ∅, φ, SAT
3  while True do
4      if status = SAT then
5          sat.add(φ_st ← next_stratum(φ_wk, φ_re))
6          φ_wk, φ_re ← φ_wk ∪ φ_st, φ_re \ φ_st
7      if (status ← sat.solve()) = SAT then
8          if φ_re = ∅ then return (cost(sat.model, φ), sat.model)
9      else
10         if (to_relax ← core_analysis(φ_wk, sat.core)) = ∅ then return (∞, ∅)
11         relaxed, B, residuals ← split_and_relax(to_relax, sat.n_vars)
12         sat.retract(to_relax)
13         sat.add(φ_rx ← relaxed ∪ (CNF(∑_{b∈B} b = 1), ∞))
14         φ_wk, φ_re ← (φ_wk \ to_relax) ∪ φ_rx, φ_re ∪ residuals
```

In the WPM1 algorithm, variable ϕ_{wk} represents the formula that contains the merged clusters (strata) so far, while ϕ_{re} represents the remaining weighted clauses from the original input instance ϕ. Whenever we solve to optimality the current instance ϕ_{wk}, i.e., the SAT solver returned a SAT answer in the last call (line 4) but $\phi_{re} \neq \emptyset$, function *next_stratum* updates variable ϕ_{st} to the new stratum (cluster) to be merged with ϕ_{wk} [7] (the working SAT instance (line 5) and variables ϕ_{wk}, ϕ_{re} are updated accordingly (line 6)). Otherwise, the SAT solver returned UNSAT in the

[6] Recall that hard clauses have weight ∞.

[7] In [10], the first call to *next_stratum* returns the cluster of all hard clauses since their representative weight is ∞

previous call, meaning that we are still optimizing the current subproblem ϕ_{wk} and need to call the SAT solver again (line 7).

If the SAT solver returns a SAT answer and all the original clauses in ϕ have been considered, i.e. $\phi_{re} = \emptyset$, then we have optimized the input instance ϕ and return its cost and an optimal model (line 8).

If the SAT solver returns an UNSAT answer, first we analyze the unsatisfiable core returned by the SAT solver (line 10) and return the soft-indicator clauses to be relaxed in variable *to_relax*, if any; otherwise, we have certified that the set of hard clauses is unsatisfiable, i.e., we return cost ∞ and an empty model.

Function *split_and_relax* (line 11) first applies the split rule to the soft-indicator clauses in *to_relax* and generates two sets, one where all the clauses are normalized to have the minimum weight, and another with the residuals of each clause with respect to the minimum weight in *to_relax*. Second, the set of clauses with the minimum weight are extended, each with an additional fresh variable and stored in the set *relaxed* as in the Fu&Malik algorithm. The new fresh variables are returned in set B.

Finally, the original set of clauses *to_relax* is retracted from the SAT solver (line 12), and the new set *relaxed* is added to the working SAT instance plus the cardinality constraint that increases the lower bound as in the Fu&Malik algorithm (line 13)[8]. In line 14, ϕ_{wk} is updated to reflect the changes in the SAT working formula, and the remaining formula ϕ_{re} is extended with the residuals generated from the application of the split rule.

As a final remark, notice that if the statements in grey boxes of the WPM1 algorithm are erased and function *next_stratum* is instructed to report sequentially, first the hard clauses and then the soft clauses, we get the original Fu&Malik algorithm.

In the context of the Covering Array Number problem, the Fu&Malik algorithm on the $PMSat^{N,t,S,lb}_{CCX}$ instance will perform a bottom-up search, i.e, the first query will correspond to the question of whether the covering array can be constructed with $k = 0$ tests, then with $k = 1$ tests, etc. This approach does not provide any intermediate upper bounds since the only query answered positively corresponds to the optimum.

However, interestingly, by considering the weighted version of the Fu&Malik algorithm, we can perform a top-down search on the Covering Array problem and provide intermediate upper bounds.

Proposition 8. The Stratified WPM1 algorithm with input $WPMSat^{N,t,S,lb}_{CCX}$ can *simulate* the CALOT algorithm (excluding lines 9 and 10).

Back to the context of covering arrays, each cluster in $WPMSat^{N,t,S,lb}_{CCX}$ would be composed of a single soft clause $(\neg u_i, w_i)$, except the cluster containing all the hard clauses. The first subproblem seen by the Stratified WPM1 algorithm encodes the query of whether one can build a covering array using N tests. The next subproblem incorporates the first soft clause $(\neg u_N, w_N)$ and encodes the query of whether one can construct the covering array using $N - 1$ tests. Notice that each $\neg u_i$ will propagate, according to CCU, the corresponding c^{i-1}_{τ} vars as in the CALOT algorithm. Notice also that every solution of a subproblem is an upper bound for the covering array.

The discussion of this section has provided insights into how to solve Covering Arrays through MaxSAT, but also into how to fix similar difficulties in other problems where MaxSAT is not yet effective enough.

[8]Notice that $(CNF(\sum_{b \in b\_vars} b = 1), \infty)$ is a set of clauses that have ∞ weight.

3.6.3 Test-based Streamliners for the $CAN(t, S)$ problem

Notice that a solution for a $CAN(t, S)$ problem can be extended to multiple solutions in the previous MaxSAT translations. This happens when $CAN(t, S) < N$, since the assignment to the x vars related to any test i with $i > CAN(t, S)$ (useless from the point of view of the $CAN(t, S)$ problem) still needs to be consistent with the X and $SUTX$ constraints. In general, notice that $SUTX$ can be NP-complete.

Lines 9 and 10 of the CALOT algorithm, as described in section 3.4, fix that problem but cannot directly be applied within MaxSAT algorithms since the solver is not aware of the $CAN(t, S)$ problem semantics.

However, we can reproduce a similar effect. At the preprocessing step, we can build a *dummy* test case v by computing a solution to S_φ (e.g. with a SAT solver) or select any of the test cases in the solution returned by the ACTS tool when computing the upper bound (see section 3.2). Then, we can state in the MaxSAT encoding that if a given test i is not part of the optimal solution (i.e., u_i is False), then the corresponding x vars are set to the value in the test case v.

$$\bigwedge_{i \in [lb+2 \ldots N]} \left(\neg u_i \rightarrow \bigwedge_{(p,v) \in v} x_{i,p,v} \right) \tag{NUX}$$

The *dummy* test case v exactly plays the role of the so-called streamliner constraints [57], which rule out some of the possible solutions but make the search of the remaining solutions more efficient.

There is yet another way to mitigate that potential bottleneck. We can indeed extend $SUTX$ clauses for test i with literal $\neg u_i$. Therefore, whenever test i is no longer in the optimal solution (i.e. u_i is False), the corresponding SUT constraints are trivially satisfied. However, in the experimental investigation, we confirmed that this option is less efficient than adding NUX clauses.

Example 14. *For the SUT in Example 5, let us assume that we use the following* dummy *test* $v = \{(Os, L), (Pl, C), (Re, K), (Or, L)\}$. *Then, the NUX encoding for* v *is:*

$$\begin{aligned}
&\neg u_{22} \rightarrow (x_{22,Os,L} \wedge x_{22,Pl,C} \wedge x_{22,Re,K} \wedge x_{22,Or,L}) \\
&\neg u_{21} \rightarrow (x_{21,Os,L} \wedge x_{21,Pl,C} \wedge x_{21,Re,K} \wedge x_{21,Or,L}) \\
&\neg u_{20} \rightarrow (x_{20,Os,L} \wedge x_{20,Pl,C} \wedge x_{20,Re,K} \wedge x_{20,Or,L}) \\
&\neg u_{19} \rightarrow (x_{19,Os,L} \wedge x_{19,Pl,C} \wedge x_{19,Re,K} \wedge x_{19,Or,L})
\end{aligned} \tag{Ex. NUX}$$

3.7 The $T(N; t, S)$ problem as Weighted Partial MaxSAT

For some applications, we may not be able to use as many test cases as the covering array number (e.g. due to budget restrictions), but we may still be interested in solving the Tuple Number problem, i.e., to determine the maximum number of covered t-tuples we can get with a test suite of fixed size.

Once again, MaxSAT technology can play an important role when SUT constraints are considered. Moreover, the size of the SAT/MaxSAT encodings for this problem are smaller than encodings for computing the Covering Array Number, since fewer tests are taken into consideration.

In the following, we show how we can modify the $Sat_{CX}^{N,t,S}$ and $Sat_{CCX}^{N,t,S}$ formulae to become Partial MaxSAT encodings of the Tuple Number problem.

The basic idea is that we need to soften the hard restriction that enforces all allowed t-tuples to be covered. To this end, we modify the SAT instance $Sat_{CX}^{N,t,S}$ as follows: First, we soften all the clauses from equation C which encode that every t-tuple τ must be covered by at least one test case, therefore allowing to violate (or relax) these constraints. For the sake of clarity, although not required for soundness, we introduce a new set of indicator variables c_τ that reify each ALO constraint in equation C by introducing the following hard constraints:

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau \leftrightarrow \bigvee_{i \in [N]} c_\tau^i) \tag{RC}$$

Then, we add the following set of soft clauses:

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau, 1). \tag{SoftC}$$

Finally, we we replace in $Sat_{CX}^{N,t,S}$ the set of constraints C (the hard constraint that forced to cover all the tuples) by the previous two sets of constraints.

Proposition 9. Let S be a SUT model and let $TPMSat_{CX}^{N,t,S}$ be $Sat_{CX}^{N,t,S}\left\{\frac{SoftC \wedge RC}{C}\right\}$. The optimal cost of $TPMSat_{CX}^{N,t,S}$ is $|\mathcal{T}_a| - T(N;t,S)$.

Remark 4. Even if $N > lb$, we cannot use fixed-tuple symmetry breaking since we do not know whether the t-tuples that we fix will lead to an optimal solution. Therefore, fixed-tuple symmetry is disabled for all the encodings in this section.

Remark 5. When computing the tuple number, we can avoid the step of detecting all forbidden tuples since the encoding remains sound, i.e., we can interchange $\mathcal{T}_a$ by $\mathcal{T}$. Notice that those c_τ vars related to forbidden tuples will always be set to False. Moreover, notice that a core-guided algorithm may potentially detect easily as many unsatisfiable cores as forbidden tuples which include just the unit soft clause that represents the forbidden tuple.

In case we want to extend $Sat_{CCX}^{N,t,S}$ to compute the tuple number, we just need to notice that the previously defined role of c_τ corresponds exactly to variable c_τ^N in $Sat_{CCX}^{N,t,S}$, so we just need to soften the hard unit clauses c_τ^N (described in CCX) with weight 1.

Proposition 10. Let S be a SUT model and let $TPMSat_{CCX}^{N,t,S}$ be $Sat_{CCX}^{N,t,S}\left\{\frac{(c_\tau^N,1)}{(c_\tau^N)}\right\}$. The optimal cost of $TPMSat_{CCX}^{N,t,S}$ is $|\mathcal{T}_a| - T(N;t,S)$.

Example 15. *We show how to build $TPMSat_{CX}^{N=22,t=2,S}$ for the SUT in Example 5.*

We must create a new variable c_τ for each value tuple in $\mathcal{T}_a$ and then replace constraint C in $SAT_{CX}^{N=22,t=2,S}$ (see Example 11) by RC (left). Finally, we have to add the $SoftC$ soft clauses (right):

$$\begin{array}{ll} c_{\tau_1} \leftrightarrow (c_{\tau_1}^1 \vee c_{\tau_1}^2 \vee \cdots \vee c_{\tau_1}^{22}) & (c_{\tau_1}, 1) \\ \vdots & \vdots \\ c_{\tau_{69}} \leftrightarrow (c_{\tau_{69}}^1 \vee c_{\tau_{69}}^2 \vee \cdots \vee c_{\tau_{69}}^{22}) & (c_{\tau_{69}}, 1) \end{array} \qquad \text{(Ex. RC and SoftC)}$$

For the $TPMSat_{CXX}^{N=22,t=2,S}$, we just have to soften, with weight 1, the set of clauses from CCX (b) in $SAT_{CCX}^{N=22,t=2,S}$ (see Example 11).

In what follows, we present two extensions.

3.7.1 Combining the $CAN(t,S)$ and $T(N;t,S)$ problems

The Covering Array and Tuple Number problems can lead to thinking about a more general formulation of the optimization problem where we want to maximize the number of covered t-tuples while minimizing the number of test cases. Notice that it will depend on the value of N with respect to the covering array number (not necessarily known a priori) whether we are, in essence, solving the covering array number or the tuple number problem.

To this end, we take the $PMSat_{CX}^{N,t,S,lb}$ encoding of the Covering Array Number problem for a SUT model S, N tests and strength t. As earlier shown in this section, we first replace the set of hard constraints C by RC and $SoftCWU$.

$$\bigwedge_{\tau \in \mathcal{T}_a} (c_\tau, |u_i| + 1). \tag{SoftCWU}$$

Notice that we prefer violating all soft clauses $(\neg u_i, 1)$ over violating a single soft clause $(c_\tau, |u_i| + 1)$. This way, we guarantee that any solution to our new Weighted Partial MaxSAT instance maximises the number of covered t-tuples while minimises the number of needed test cases.

Proposition 11. If $N \geq CAN(t,S)$, the optimal cost of the Weighted Partial MaxSAT instance $PMSat_{CX}^{N,t,S,lb}\left\{\frac{SoftCWU \wedge RC}{C}\right\}$ is $CAN(t,S) - (lb+1) + (|\mathcal{T}_a| - T(N;t,S)) \cdot (|u_i|+1)$, otherwise it is $N - (lb+1) + (|\mathcal{T}_a| - T(N;t,S)) \cdot (|u_i|+1)$. [9]

The same idea can be applied to $PMSat_{CCX}^{N,t,S,lb}$ by softening the unit hard clauses (c_τ^N) in equation (b) from CCX with weight $|u_i|+1$. Here, it is important to recall the discussion in section 3.1 on the need of equation (c) in CCX. The other, perhaps more natural, alternative was to replace equation (c) in CCX by $\bigwedge_{\tau \in \mathcal{T}_a} \neg c_\tau^0$. The problem arises when, in an optimal solution, τ is not covered, what also implies that (c_τ^N) is False. Notice that we need to satisfy all clauses related to τ in CCX but, in order to do that, we need to set all c_τ^i vars to False. This may not be compatible with equation CCU (clauses of the form $\neg c_\tau^{i-1} \rightarrow u_i$) when some test i is discarded to be in the solution and variable u_i is set to False, since UP will derive in CCU that c_τ^{i-1} is True. In this case, a contradiction is reached. On the other hand, as discussed in section 3.1, equation (c) allows to set all c_τ^i vars to True when (c_τ^N) is False and trivially satisfy all clauses in CCX related to τ.

Proposition 12. If $N \geq CAN(t,S)$, the optimal cost of the Weighted Partial MaxSAT instance $PMSat_{CCX}^{N,t,S,lb}\left\{\frac{SoftCWU}{(c_\tau^N)}\right\}$ is $CAN(t,S) - (lb+1) + (|\mathcal{T}_a| - T(N;t,S)) \cdot (|u_i|+1)$, otherwise it is $N - (lb+1) + (|\mathcal{T}_a| - T(N;t,S)) \cdot (|u_i|+1)$. [9]

3.7.2 The $CAN(t,S)$ problem with Relaxed Tuple Ratio Coverage as MaxSAT

We can tackle other realistic settings where we still want to use the minimum number of tests, but there is no need to achieve a 100% ratio of covered t-tuples (mandatory per definition in Covering Arrays). Notice that the last tests that shape the covering array number tend to cover very few not yet covered t-tuples. Therefore, if

[9] Notice that if $N \geq CAN(t,S)$, then $|\mathcal{T}_a| - T(N;t,S)$ is 0. However, we keep this expression in case we want to interchange $\mathcal{T}_a$ by $\mathcal{T}$. i.e., if we do not prefilter the forbidden tuples (see Remark 5).

these tests are expensive enough in our setting, we may consider relaxing the ratio coverage and skip these tests.

The mentioned problem can be encoded by replacing the previously soft constraints on the c_τ vars with a hard cardinality constraint on the minimum number of t-tuples to be covered as follows:

$$\sum_{\tau \in \mathcal{T}_a} c_\tau \geq \lceil |\mathcal{T}_a| \cdot rt \rceil \qquad (CCard)$$

where rt is the ratio of allowed t-tuples that we want to cover. Notice that, for efficiency reasons, $CCard$ can be also described as $\sum_{\tau \in \mathcal{T}_a} \neg c_\tau \leq \lceil |\mathcal{T}_a| \cdot (1 - rt) \rceil$.

Remark 6. With this formulation, we cannot use the fixed-tuples symmetry breaking since we do not know whether we will require at least lb tests to cover the specified ratio of allowed t-tuples.

Proposition 13. Let $RTPMSat_{CCX}^{N,t,S,rt}$ be $PMSat_{CCX}^{N,t,S,lb=0}\left\{\frac{CCard}{(c_\tau^N)}\right\}$. The optimal cost of $RTPMSat_{CCX}^{N,t,S,rt}$ is the minimum N' such that $T(N', t, S) \geq \lceil |\mathcal{T}_a| \cdot rt \rceil$.

3.8 Incomplete MaxSAT Algorithms for the $T(N; t, S)$ problem

As argued earlier, if certifying optimality is not a requirement and we are just interested in obtaining a good suboptimal solution in a reasonable amount of time, we can apply incomplete MaxSAT algorithms on the encodings of the Tuple Number problem described in the previous section. Additionally, in this section, we present a new incomplete algorithm to compute suboptimal solutions for the Tuple Number problem.

3.8.1 MaxSAT based Incremental Test Suite Construction

A way to reduce the search space of any constraint problem is to add the so-called streamliner constraints [57]. We recall that these constraints rule out some of the possible solutions but make the search for the remaining solutions more efficient. However, in practice, streamliners can rule out all the solutions.

In our context, the streamliner constraints correspond to a set of tests that we think have the potential to be part of optimal solutions. By fixing these tests, we generate a new covering array problem, easier to solve, but whose Covering Array Number can be greater than or equal to that of the original covering array, because we may have missed all the optimal solutions. We iterate this process until all t-tuples get covered. To select the k candidate test to be fixed at each iteration, we solve the Tuple Number problem restricted to length k.

In the context of the Tuple Number problem, this iterative process of fixing tests should not only finish when all t-tuples have been covered but also when the requested N tests have been fixed.

To that end, here we combine a greedy iterative approach with the SAT-based MaxSAT approaches from section 3.7 in the IncrementalCA algorithm.

In this algorithm, we begin with the remaining tuples to cover $\mathcal{T}_r$, initially assigned to allowed tuples $\mathcal{T}_a$, as well as an empty test suite Y (line 2). Then, we first check how many tests should be encoded; the minimum between the tests in iteration N_i and the remaining number of tests left to complete the test suite, $N - |\mathrm{Y}|$ (line

Algorithm IncrementalCA: MaxSAT based Incremental Test Suite Construction

1 **Input:** *SUT model S, Tests N_i per iteration, SAT-based MaxSAT solver msat*
2 $\mathcal{T}_r, \mathrm{Y} \leftarrow \mathcal{T}_a, \emptyset$
3 **while** $\mathcal{T}_r \neq \emptyset$ **and** $|\mathrm{Y}| < N$ **do**
4 $\quad N' \leftarrow min(N_i, N - |\mathrm{Y}|)$
5 $\quad msat.add\left(TPMSat_{CCX}^{N',t,S}\right)$
6 $\quad msat.solve()$
7 $\quad v \leftarrow$ tests from $msat.model$
8 $\quad \mathrm{Y} \leftarrow \mathrm{Y} \cup v$
9 $\quad \mathcal{T}_r \leftarrow \mathcal{T}_r \setminus \{\tau \mid v \models \tau\}$
10 **return** Y

4), storing the result into N'. Next, we solve the Tuple Number problem for these N' tests, encoded as a $TPMSat_{CCX}^{N',t,S}$ formula (lines 5, 6) from section 3.7. We extract the model from the MaxSAT solver, interpreting it into newly found test cases v (line 7). Then, those new tests are added to test suite Y (line 8). Finally, the tuples covered by these new test cases are removed from $\mathcal{T}_r$ (line 9). This iteration is repeated until no more tuples are left in $\mathcal{T}_r$ or we have reached the requested N test cases (line 3), in which case we return the constructed test suite Y (line 10).

3.9 Experimental Evaluation

In this section, we report on an extensive experimental investigation conducted to assess the approaches proposed in the preceding sections. We start by defining the benchmarks, which include 28 industrial, real-world or real-life instances and 30 crafted instances, and the algorithms involved in the evaluation.

We contacted the authors of [101] and [100] to obtain the benchmarks used in their experiments. In particular, the available benchmarks are: (i) Cohen et al. [38], with 5 real-world and 30 artificially generated (crafted) covering array problems; (ii) Segall et al. [96], with 20 industrial instances; (iii) Yu et al. [102], with two real-life systems reported by ACTS users; and (iv) Yamada et al. [100], with an industrial instance named "Company_B".

Table 3.1 provides information about the System Under Test of each instance, where S_P is the number of parameters and their domain (e.g. the meaning of $2^{29}3^1$ in instance 7 is that the instance contains 29 parameters of domain 2 and 1 parameter of domain 3); S_φ is the number of SUT constraints and their sizes (e.g. the meaning of $2^{13}3^2$ in instance 7 is that the instance contains 13 constraints of size 2 and 2 constraints of size 3); and # *lits* $CNF(S_\varphi)$ is the number of literals of the CNF representation of S_φ (i.e. the sum of the sizes of all clauses).

Table 3.1 also reports, for $t = 2$, the following data: ub^{ACTS}, which indicates the upper bound returned by the ACTS tool (see section 3.2); $ub^{\simeq}$, which is the best known upper bound (a star indicates that it is optimal, i.e., $CAN(2, S)$); lb, which reports the lower bound (computed as in section 3.2); and $|\mathcal{T}_a|$ and $|\mathcal{T}_f|$, which report the number of allowed and forbidden tuples, respectively.

Finally, we also show, for the $PMSat_{CCX}^{N,t=2,S,lb}$ encoding of each instance, the following information: # *vars*, which is the number of variables used by this encoding;

clauses, which is the number of clauses; # *lits*, which is the number of literals; and *size (MB)*, which is the file size of the WCNF formula in MB.

Notice that we focus on $t = 2$ strength coverage.

	System Under Test (SUT)			Bounds for $t = 2$					$PMSat_{CCX}^{N,t=2,S,lb}$			
Instance	S_P	S_φ	# lits $CNF(S_\varphi)$	ub^{ACTS}	$ub^{\simeq}$	lb	$\lvert\mathcal{T}_a\rvert$	$\lvert\mathcal{T}_f\rvert$	# vars	# clauses	# lits	size (MB)
Cohen et al. [38]												
1	$2^{86}3^{3}4^{1}5^{5}6^{2}$	$2^{20}3^{3}4^{1}$	53	48	37	35	23876	474	1158588	2620675	7463282	60.01
2	$2^{86}3^{3}4^{3}5^{1}6^{1}$	$2^{19}3^{3}$	47	32	30*	29	20331	237	657890	1371738	3984183	29.91
3	$2^{27}4^{2}$	$2^{9}3^{1}$	21	19	18*	15	1838	14	36217	79008	222390	1.47
4	$2^{51}3^{4}4^{2}5^{1}$	$2^{15}3^{2}$	36	22	20*	19	7530	386	168852	358291	1025536	7.33
5	$2^{155}3^{7}4^{3}5^{5}6^{4}$	$2^{32}3^{6}4^{1}$	86	54	45	35	76259	73	4142574	9720622	27451121	236.74
6	$2^{73}4^{3}6^{1}$	$2^{26}3^{4}$	64	25	24*	23	11382	1878	289001	597814	1730859	12.72
7	$2^{29}3^{1}$	$2^{13}3^{2}$	32	12	9	5	1567	231	19566	49758	132435	0.85
8	$2^{109}3^{2}4^{2}5^{3}6^{3}$	$2^{32}3^{4}4^{1}$	80	47	36*	35	33680	1098	1597165	3590459	10247230	84.50
9	$2^{57}3^{1}4^{1}5^{1}6^{1}$	$2^{30}3^{7}$	81	22	20*	19	6835	1720	153584	325984	932515	6.63
10	$2^{130}3^{6}4^{5}5^{2}6^{4}$	$2^{40}3^{7}$	101	47	41	35	52659	2029	2493173	5608369	16010703	135.34
11	$2^{84}3^{4}4^{2}5^{2}6^{4}$	$2^{28}3^{4}$	68	47	39	35	23636	707	1123311	2523897	7200149	57.70
12	$2^{136}3^{4}4^{3}5^{1}6^{3}$	$2^{23}3^{4}$	58	43	36*	35	49522	978	2144718	4675267	13461992	108.23
13	$2^{124}3^{4}4^{1}5^{2}6^{2}$	$2^{22}3^{4}$	56	40	36*	35	38862	1701	1567084	3319517	9632256	75.77
14	$2^{81}3^{5}4^{3}6^{3}$	$2^{13}3^{2}$	32	39	36*	35	20544	618	810618	1697204	4936072	37.18
15	$2^{50}3^{4}4^{1}5^{2}6^{1}$	$2^{20}3^{2}$	46	32	30*	29	8388	155	273410	569181	1650514	12.10
16	$2^{81}3^{4}2^{6}1^{1}$	$2^{30}3^{4}$	72	25	24*	23	14600	2303	370051	765960	2218422	16.44
17	$2^{128}3^{3}4^{2}5^{1}6^{3}$	$2^{25}3^{4}$	62	41	36*	35	43390	66	1792402	3835891	11100545	88.14
18	$2^{127}3^{2}4^{4}5^{6}6^{2}$	$2^{23}3^{4}4^{1}$	62	52	41	35	50128	28	2625808	6092947	17250882	146.38
19	$2^{172}3^{9}4^{9}5^{3}6^{4}$	$2^{38}3^{5}$	91	51	43	35	98778	114	5064366	11694341	33170488	287.31
20	$2^{138}3^{4}4^{5}5^{4}6^{7}$	$2^{42}3^{6}$	102	60	54	35	64620	3320	3903864	9411047	26386102	227.62
21	$2^{76}3^{3}4^{2}5^{1}6^{3}$	$2^{40}3^{6}$	98	39	36*	35	15442	2742	610938	1279170	3717471	27.90
22	$2^{72}3^{4}4^{1}6^{2}$	$2^{20}3^{2}$	46	37	36*	35	13405	1181	503127	1028139	3008516	22.48
23	$2^{25}3^{1}6^{1}$	$2^{13}3^{2}$	32	14	12*	11	1495	173	21856	47740	132915	0.85
24	$2^{110}3^{2}5^{3}6^{4}$	$2^{25}3^{4}$	62	48	41	35	34204	570	1656252	3748659	10679658	88.30
25	$2^{118}3^{6}4^{2}5^{2}6^{6}$	$2^{23}3^{4}4^{1}$	59	52	49	35	46968	52	2461280	5710735	16167454	136.81
26	$2^{87}3^{1}4^{3}5^{4}$	$2^{28}3^{4}$	68	34	26	24	20921	667	719347	1643461	4647485	36.52
27	$2^{55}3^{2}4^{2}5^{1}6^{2}$	$2^{17}3^{3}$	43	37	36*	35	9714	43	365524	746797	2183919	16.18
28	$2^{167}3^{16}4^{2}5^{3}6^{6}$	$2^{31}3^{6}$	80	57	50	35	96599	74	5535861	13181074	37087871	322.33
29	$2^{134}3^{7}5^{3}$	$2^{19}3^{3}$	47	29	25*	24	45839	32	1338905	2899941	8321499	64.34
30	$2^{73}3^{4}3^{3}$	$2^{31}3^{4}$	74	22	16*	15	12453	1308	277976	640938	1792681	13.16
apache	$2^{158}3^{8}4^{5}5^{1}6^{1}$	$2^{33}1^{4}2^{5}1^{1}$	22	33	30*	29	66927	3	2221926	4701044	13619419	109.16
bugzilla	$2^{49}3^{1}4^{2}$	$2^{4}3^{1}$	11	19	16*	15	5818	4	112768	247130	697953	4.82
gcc	$2^{189}3^{10}$	$2^{37}3^{3}$	83	23	15	8	82770	39	1913568	5063264	13685896	112.78
spins	$2^{13}4^{5}$	2^{13}	26	26	19*	15	979	13	27050	64498	177169	1.23
spinv	$2^{42}3^{2}4^{11}$	$2^{47}3^{2}$	100	45	33	15	8741	56	401069	1063265	2888090	22.53
Segall et al. [96]												
Banking1	$3^{4}4^{1}$	5^{112}	560	15	13*	11	102	0	1938	5864	19573	0.11
Banking2	$2^{14}4^{1}$	2^{3}	6	11	10*	7	473	3	5591	12845	34672	0.21
CommProtocol	$2^{10}7^{1}$	$2^{10}3^{10}4^{12}5^{24}$ $6^{30}7^{30}8^{12}$	704	19	16*	13	285	35	6047	15914	50363	0.29
Concurrency	2^{5}	$2^{4}3^{1}5^{2}$	21	6	5*	3	36	4	278	667	1686	0.01
Healthcare1	$2^{6}3^{2}5^{1}6^{1}$	$2^{3}3^{18}$	60	30	30*	29	361	8	12090	24661	70619	0.44
Healthcare2	$2^{5}3^{6}4^{1}$	$2^{1}3^{6}5^{18}$	110	16	14	11	466	1	8212	18853	52268	0.31
Healthcare3	$2^{16}3^{6}4^{5}5^{1}6^{1}$	2^{31}	62	38	34*	29	3092	59	121950	271023	768538	5.38
Healthcare4	$2^{13}3^{12}4^{6}5^{2}6^{1}7^{1}$	2^{22}	44	49	46*	41	5707	38	287980	619634	1783211	13.14
Insurance	$2^{6}3^{1}5^{1}6^{2}11^{1}13^{1}17^{1}31^{1}$	-	0	527	527*	526	4573	0	2509047	5009492	14863678	122.95
NetworkMgmt	$2^{2}4^{1}5^{3}10^{2}11^{1}$	2^{20}	40	112	110*	109	1228	20	148402	301059	877206	6.28
ProcessorComm1	$2^{3}3^{6}4^{6}$	2^{13}	26	29	21	15	1058	13	32957	80475	219601	1.54
ProcessorComm2	$2^{3}3^{12}4^{8}5^{2}$	$1^{4}2^{121}$	246	32	25*	24	2525	854	85287	193248	541399	3.73
Services	$2^{3}3^{4}5^{2}8^{2}10^{2}$	$3^{386}4^{2}$	1166	106	100*	99	1819	16	204692	460866	1346965	9.78
Storage1	$2^{1}3^{1}4^{1}5^{1}$	4^{95}	380	17	17*	14	53	18	1294	4270	13468	0.07
Storage2	$3^{4}6^{1}$	-	0	18	18*	17	126	0	2826	5652	15552	0.09
Storage3	$2^{9}3^{1}5^{3}6^{1}8^{1}$	$2^{38}3^{10}$	106	50	50*	39	1020	120	54810	122009	344328	2.47
Storage4	$2^{5}3^{7}4^{1}5^{2}6^{2}7^{1}10^{1}13^{1}$	2^{24}	48	136	130*	129	3491	24	495046	1012862	2970799	22.45
Storage5	$2^{5}3^{8}5^{3}6^{2}8^{1}9^{1}10^{2}11^{1}$	2^{151}	302	218	215	109	5342	246	1206084	3020149	8366680	72.16
SystemMgmt	$2^{5}3^{4}5^{1}$	$2^{13}3^{4}$	38	17	15*	14	310	14	5935	12813	35376	0.21
Telecom	$2^{5}3^{1}4^{2}5^{1}6^{1}$	$2^{11}3^{1}4^{9}$	61	32	30*	29	440	11	15650	32761	93262	0.59
Yu et al. [102]												
RL-A	$2^{5}3^{4}4^{7}5^{4}6^{5}7^{4}8^{1}12^{3}$	$1^{12}2^{491}3^{345}$	2029	155	153	143	7066	7156	1142671	2491775	7220414	59.32
RL-B	$2^{8}3^{2}4^{3}5^{3}6^{1}9^{1}$ $10^{1}12^{2}14^{3}20^{1}24^{1}37^{1}$	$1^{8}2^{1127}3^{277}$ $4^{1755}5^{1064}6^{2048}$	27721	767	727	519	17018	5597	13365222	35733711	109278283	1026.60
Yamada et al. [100]												
Company2	$2^{6}3^{4}8^{4}$	$1^{2}2^{35}3^{89}4^{54}5^{34}$ $6^{207}3^{4}8^{169}4^{}$	1247	81	72	55	1149	261	100546	252543	744203	5.35

TABLE 3.1: General information of all benchmarks used.

Regarding existing tools for solving Mixed Covering Arrays with Constraints, the main tool we compare with is CALOT [101]. Unfortunately, CALOT is not available from the authors but we did our best to reproduce it (see section 3.4), showing our experimental investigation that the results are consistent with those of [101]. Our implementation of CALOT and all algorithms presented in this chapter can be found in `http://hardlog.udl.cat/static/doc/inc-maxsat-ct/html/index.html`,

which we think is also a nice contribution for both the combinatorial testing and satisfiability communities.

Since all the algorithms presented in this chapter are built on top of a SAT solver, we compared, when possible, all the algorithms with the same underlying SAT solver. That is not the case in [101], which may lead to flawed conclusions. In our experimental investigation we choose Glucose (version 4.1) [16], as most of the state-of-the-art MaxSAT solvers are built on top of it.

We also use the ACTS tool [34] to compute fast and good enough upper bounds of the Covering Array Number problem, although it is not competitive with SAT-based approaches.

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. Unless otherwise stated, all the experiments were executed with a timeout of 2h and a memory limit of 18GB. To mitigate the impact of randomness we executed all the algorithms using five different seeds for each instance.

The rest of the experimental section is organized as follows. Regarding the Covering Array Number, in subsection 3.9.1, we compare the CALOT algorithm with the MaxSAT encodings and SAT-based MaxSAT approaches described in sections 3.5 and 3.6. Regarding the Tuple Number problem, in subsection 3.9.2, we evaluate the complete and incomplete MaxSAT algorithms on the encoding described in section 3.7. Then, in subsection 3.9.3, we evaluate the incomplete approach for computing the Tuple Number described in section 3.8.

3.9.1 SAT-based MaxSAT approaches for the Covering Array Number problem

In this experiment, we compare the performance of state-of-the-art SAT-based MaxSAT solvers with the CALOT algorithm described in section 3.4. We hypothesise that since these SAT-based MaxSAT algorithms, once executed on the suitable MaxSAT encodings, can *simulate* the behaviour of the CALOT algorithm (see Propositions 6 and 8) but the opposite is not true, MaxSAT algorithms may perform similarly or outperform the CALOT algorithm. This hypothesis would contradict the findings in [101], where it was reported that the CALOT algorithm clearly dominates the MaxSAT-based approach in [14]. If our hypothesis is correct, MaxSAT approaches for solving the Covering Array Number problem would be put back on the agenda. We focus in anytime algorithms that must be able to report suboptimal solutions [10].

Solvers: The CALOT algorithm (described in section 3.4) and the model-guided Linear SAT-based MaxSAT algorithm Linear (described in section 3.6) were implemented on top of the OptiLog [7] python framework for SAT solving. This framework includes python bindings for several state-of-the-art SAT solvers and the python binding to the PBLib [78].

We additionally tested several complete and incomplete algorithms from the MaxSAT Evaluation 2020 [23]. From complete MaxSAT solvers we tested MaxHS [21], EvalMaxSAT [20], RC2 [67] and maxino [3]. We only report results for RC2 and one seed[11], as this was the complete solver that reported better results. MaxHS obtained the best results for 2 of the tested instances, but we decided to exclude it from

[10] We adapted RC2 MaxSAT solver to report suboptimal solutions when applying the stratified strategy (see section 3.6)

[11] Unfortunately *RC2* MaxSAT solver does not allow to specify a seed.

the comparison since it cannot report upper bounds for most of the instances and it uses another underlying SAT solver than Glucose41.

Regarding incomplete MaxSAT algorithms we tested Loandra [26], tt-open-wbo-inc [86] and SatLike [74]. We report results for Loandra and tt-open-wbo-inc as SatLike crashed in some of the tested instances.

MaxSAT encodings: We report results on $PMSat^{N,t,S,lb}_{CCX}$ and the weighted version $WPMSat^{N,t,S,lb}_{CCX}$ using a linear increase for the weights ($w_i = i - (lb+2) + 1$, see equation $WSoftU$ in section 3.5). We found that $WPMSat^{N,t,S,lb}_{CCX}$ with the linear and exponential increase ($w_i = 2^{i-(lb+2)}$) lead to the same performance, but the exponential increase represented a problem for some MaxSAT solvers when i was high enough.

We further tested the three different alternatives for equation (a) from CCX, where two reported good results. The first one is the original (a) equation shown in section 3.1, $(c^i_\tau \rightarrow c^{i-1}_\tau \vee x_{i,p,v})$, which we will refer to as a.0. The second one is the variation $(c^i_\tau \rightarrow c^{i-1}_\tau \vee x_{i,p,v}) \wedge (c^i_\tau \leftarrow c^{i-1}_\tau)$, which we will refer to as a.1.

Results: Table 3.2 shows the results of our experimentation. For each row and solver column, we give the average size of the minimum MCAC (out of the 5 executions per instance) and the average runtime. Bold values represent the best results. In case there are ties in size, the best time is marked. Sizes that have a star represent that the optimum has been certified in at least one of the five seeds executed for the current benchmark instance.

Table 3.3 aggregates the information presented in Table 3.2 to analyze the dominance relations among approaches. In particular, we show for each row the number of wins (W) and loses (L) with respect to each of the approaches in columns, for both size and run times. We consider that if algorithm A finds a smaller MCAC than B, then A also needs less runtime than B. In this sense, we will say that an approach outperforms another if it provides a strictly better solution within the given timeout or finds the same best suboptimal solution faster. For example, in the *ACTS* row we found that it obtains worse sizes than CALOT CCX a.0 in 52 instances (0 W, 52 L in column *size*), better runtimes in 2 and worse runtimes in 56 (2 W, 56 L in column *time*).

We observe how both *tt-open-wbo-inc* and *loandra* outperform the results obtained by *CALOT*, improving the sizes in more than 10 of the 58 available instances and, in the case of *tt-open-wbo-inc*, we also improve runtimes in more than 40 instances. This confirms our hypothesis that MaxSAT approaches can *simulate* and even improve the results obtained by the *CALOT* algorithm.

Regarding the different variations of the *CCX* encoding, we notice that for *tt-open-wbo-inc* and *loandra*, variation a.1 slightly improves results obtained by the original variation a.0. In particular, we observe that *tt-open-wbo-inc* with this specific encoding obtains the best size[12] in instance *RL-B* (727), while algorithm *CALOT* reports a size of 760. However, this behaviour of the encoding a.1 is not observed in algorithm *CALOT*, as in this case, the best variation of equation (a) seems to be a.0. These results suggest that in case we use a new MaxSAT solver we should not discard at front any encoding variation.

For *RC2* and *linear* approaches we can observe clear differences among them when applying the $PMSat^{N,t,S,lb}_{CCX}$ encoding, as *linear* obtains better sizes and times in 21 and 57 instances respectively. These results show that, for the Covering Array Number problem, it is more effective to perform a search that incrementally refines the upper bound as the linear approach does (see section 3.6). However, we observe

[12]To the best of our knowledge this is the best known upper bound for $t = 2$ for this instance.

Instance	ACTS size	ACTS time	CALOT CCX a.0 size	CALOT CCX a.0 time	CALOT CCX a.1 size	CALOT CCX a.1 time	RC2-B CCX a.0 size	RC2-B CCX a.0 time	RC2-B CCX a.0 wpm size	RC2-B CCX a.0 wpm time	linear CCX a.0 size	linear CCX a.0 time	loandra CCX a.0 size	loandra CCX a.0 time	loandra CCX a.1 size	loandra CCX a.1 time	tt-open-wbo-inc CCX a.0 size	tt-open-wbo-inc CCX a.0 time	tt-open-wbo-inc CCX a.1 size	tt-open-wbo-inc CCX a.1 time	tt-open-wbo-inc CCX a.0 wpm size	tt-open-wbo-inc CCX a.0 wpm time	tt-open-wbo-inc CCX a.1 wpm size	tt-open-wbo-inc CCX a.1 wpm time
Cohen et al. [38]																								
1	48.00	0.83	38.00	1688.39	38.00	1470.88	-1.00	-1.00	37.00	7146.52	37.00	5367.02	37.00	1718.97	**37.00**	**426.06**	37.00	902.46	38.00	133.26	37.00	4724.61	38.00	1196.04
2	32.00	1.08	30.00*	2.80	30.00*	3.57	30.00*	9.85	30.00*	10.81	30.00*	2.91	30.00*	7.19	30.00*	7.68	**30.00**	**0.81**	30.00	1.42	30.00	0.92	30.00	1.70
3	19.00	1.21	18.00*	0.09	18.00*	0.12	18.00*	0.59	18.00*	0.73	18.00*	0.13	18.00*	0.36	18.00*	0.47	18.00	0.06	18.00	0.08	**18.00**	**0.05**	18.00	0.07
4	22.00	1.04	20.00*	0.52	20.00*	0.95	20.00*	2.58	20.00*	3.19	20.00*	0.74	20.00*	1.34	20.00*	2.24	20.00	0.30	**20.00**	**0.29**	20.00	0.31	20.00	0.45
5	54.00	1.31	47.00	1955.92	48.00	241.15	-1.00	-1.00	48.00	378.46	47.00	1693.29	46.00	1629.29	**45.80**	**1795.81**	46.00	1624.87	46.00	1348.31	46.20	1803.99	46.00	4655.60
6	25.00	1.11	24.00*	1.18	24.00*	1.46	24.00*	4.46	24.00*	5.47	24.00*	1.48	24.00*	2.98	24.00*	3.51	**24.00**	**0.33**	24.00	0.48	24.00	0.34	24.00	0.47
7	12.00	1.13	9.00	0.06	9.00	0.08	-1.00	-1.00	9.00	0.61	9.00	0.13	9.00	30.98	9.00	30.57	9.00	0.06	9.00	0.06	**9.00**	**0.05**	9.00	0.06
8	47.00	1.01	38.00	1403.51	37.00	4727.41	-1.00	-1.00	37.00	3407.87	38.00	547.77	36.80*	2849.90	37.00	3731.63	37.00	1178.58	**36.60**	**2948.62**	37.00	2333.83	37.00	5133.82
9	22.00	1.21	20.00*	0.57	20.00*	0.84	20.00*	2.36	20.00*	2.66	20.00*	0.49	20.00*	1.22	20.00*	2.30	**20.00**	**0.18**	20.00	0.27	20.00	0.19	20.00	0.41
10	47.00	1.28	44.00	6860.63	44.20	5250.31	-1.00	-1.00	44.00	4759.95	44.00	1343.51	41.80	1235.36	41.00	1891.73	41.00	2720.18	41.00	970.10	41.00	5823.09	**41.00**	**641.10**
11	47.00	1.26	41.00	2641.09	41.00	4167.69	-1.00	-1.00	42.00	6172.69	42.00	1916.85	39.00	6308.22	40.00	4323.47	40.00	741.96	40.00	1086.24	**39.00**	**1628.02**	39.00	2892.01
12	43.00	1.29	36.00*	14.13	36.00*	24.01	36.00*	36.81	36.00*	49.94	36.00*	20.66	36.00*	36.16	36.00*	42.45	**36.00**	**8.18**	36.00	10.46	36.00	10.75	36.00	11.37
13	40.00	1.15	36.00*	6.78	36.00*	9.80	36.00*	24.42	36.00*	29.91	36.00*	6.73	36.00*	14.91	36.00*	25.04	36.00	3.46	36.00	3.31	**36.00**	**3.23**	36.00	3.60
14	39.00	0.83	36.00*	3.76	36.00*	5.25	36.00*	13.81	36.00*	15.56	36.00*	4.80	36.00*	8.84	36.00*	11.51	36.00	1.89	36.00	2.48	**36.00**	**1.51**	36.00	4.28
15	32.00	1.05	30.00*	1.02	30.00*	1.49	30.00*	4.73	30.00*	4.88	30.00*	1.30	30.00*	2.91	30.00*	3.70	**30.00**	**0.47**	30.00	0.66	30.00	0.90	30.00	0.77
16	25.00	0.84	24.00*	1.36	24.00*	1.94	24.00*	5.59	24.00*	6.60	24.00*	1.65	24.00*	4.30	24.00*	4.46	**24.00**	**0.56**	24.00	0.60	24.00	0.57	24.00	0.62
17	41.00	1.17	36.00*	10.16	36.00*	13.60	36.00*	30.57	36.00*	35.75	36.00*	10.98	36.00*	22.72	36.00*	31.18	**36.00**	**5.88**	36.00	7.48	36.00	7.31	36.00	6.94
18	52.00	1.30	42.00	128.65	42.00	201.42	-1.00	-1.00	42.00	177.28	42.00	202.10	41.00	637.62	41.00	734.99	**41.00**	**134.63**	41.00	626.83	41.00	2943.99	41.00	1871.62
19	51.00	1.43	46.00	178.56	45.00	118.23	-1.00	-1.00	46.00	265.76	46.00	128.89	44.00	2056.11	**43.80**	**2638.11**	44.00	755.22	44.00	756.79	44.00	2892.85	44.00	578.78
20	60.00	1.53	57.00	129.90	57.00	224.31	-1.00	-1.00	57.00	236.47	56.00	1092.85	54.80	1422.09	**54.60**	**1736.92**	55.00	32.47	55.00	500.42	55.00	437.94	55.00	176.33
21	39.00	1.33	36.00*	2.59	36.00*	3.86	36.00*	9.99	36.00*	9.95	36.00*	3.40	36.00*	7.08	36.00*	8.88	**36.00**	**1.06**	36.00	1.38	36.00	1.72	36.00	1.94
22	37.00	1.08	36.00*	1.75	36.00*	2.70	36.00*	7.74	36.00*	8.57	36.00*	2.17	36.00*	5.25	36.00*	6.78	**36.00**	**0.65**	36.00	0.88	36.00	0.72	36.00	1.04
23	14.00	0.75	12.00*	0.05	12.00*	0.09	12.00*	0.42	12.00*	0.41	12.00*	0.08	12.00*	0.22	12.00*	0.30	**12.00**	**0.03**	12.00	0.05	12.00	0.06	12.00	0.09
24	48.00	1.40	44.40	4172.55	45.00	185.06	-1.00	-1.00	44.00	41.40	45.00	13.58	42.00	1693.19	**41.00**	**4304.36**	42.00	1465.17	43.00	59.40	42.00	884.17	42.00	926.51
25	52.00	1.28	52.00	100.56	52.00	117.86	-1.00	-1.00	-1.00	-1.00	52.00	423.91	50.00	2069.85	**49.80**	**1133.70**	51.00	136.11	50.00	2101.71	50.00	5892.75	50.00	263.99
26	34.00	1.37	27.00	49.99	27.00	31.54	-1.00	-1.00	27.00	112.13	27.00	551.57	27.00	103.46	27.00	57.66	26.00	6740.38	**26.00**	**790.74**	27.00	210.04	27.00	141.21
27	37.00	1.32	36.00*	1.14	36.00*	1.68	36.00*	5.35	36.00*	6.70	36.00*	1.33	36.00*	2.96	36.00*	3.56	**36.00**	**0.39**	36.00	0.61	36.00	0.45	36.00	0.62
28	57.00	1.54	53.00	95.96	53.00	78.94	-1.00	-1.00	53.00	330.79	53.00	120.36	51.20	333.93	51.00	1236.07	51.00	3112.51	51.00	607.78	52.00	61.99	**49.00**	**3329.73**
29	29.00	1.15	25.00*	5.63	25.00*	8.77	25.00*	20.73	25.00*	26.38	25.00*	7.81	25.00*	12.69	25.00*	17.48	**25.00**	**3.05**	25.00	4.02	25.00	3.55	25.00	4.63
30	22.00	0.81	16.00*	1.38	16.00*	1.75	16.00*	4.53	16.00*	5.98	16.00*	1.56	16.00*	3.57	16.00*	3.94	**16.00**	**0.68**	16.00	0.83	16.00	0.77	16.00	1.08
apache	33.00	1.25	30.00*	8.96	30.00*	13.68	30.00*	33.44	30.00*	38.18	30.00*	11.28	30.00*	21.65	30.00*	32.14	**30.00**	**5.17**	30.00	6.70	30.00	5.82	30.00	5.57
bugzilla	19.00	1.03	16.00*	0.35	16.00*	0.61	16.00*	1.72	16.00*	1.79	16.00*	0.36	16.00*	0.89	16.00*	1.19	**16.00**	**0.14**	16.00	0.19	16.00	0.14	16.00	0.46
gcc	23.00	0.93	15.00	13.09	15.00	19.06	-1.00	-1.00	15.00	47.85	15.00	22.14	15.00	63.99	15.00	68.18	**15.00**	**9.44**	15.00	12.76	15.00	11.82	15.00	29.62
spins	26.00	1.22	**19.00***	**0.13**	19.00*	0.22	19.00*	0.56	19.00*	0.70	19.00*	0.16	19.00*	0.28	19.00*	0.28	19.00	0.13	19.00	0.21	19.00	0.20	19.00	0.21
spinv	45.00	0.84	33.00	5.28	32.00	188.97	-1.00	-1.00	32.00	55.75	33.00	21.39	32.00	61.37	32.00	97.82	**32.00**	**26.18**	32.00	82.41	32.00	53.36	32.00	199.38
Segall et al. [96]																								
Banking1	15.00	0.85	**13.00***	**0.01**	13.00*	0.01	13.00*	0.20	13.00*	0.29	13.00*	0.08	13.00*	0.05	13.00*	0.05	13.00	0.09	13.00	0.08	13.00	0.06	13.00	0.06
Banking2	11.00	0.46	**10.00***	**0.01**	10.00*	0.02	10.00*	25.56	10.00*	22.95	10.00*	0.03	10.00*	19.30	10.00*	10.62	10.00	0.02	10.00	0.02	10.00	0.02	10.00	0.02
CommProtocol	19.00	1.05	**16.00***	**0.02**	16.00*	0.03	16.00*	0.19	16.00*	0.22	16.00*	0.03	16.00*	0.15	16.00*	0.09	16.00	0.05	16.00	0.03	16.00	0.02	16.00	0.02
Concurrency	6.00	0.49	**5.00***	**0.00**	5.00*	0.00	5.00*	0.08	5.00*	0.08	5.00*	0.01	5.00*	0.03	5.00*	0.03	5.00	0.03	5.00	0.01	5.00	0.01	5.00	0.01
Healthcare1	30.00	0.55	30.00*	0.03	30.00*	0.03	30.00*	0.24	30.00*	0.25	30.00*	0.05	30.00*	0.13	30.00*	0.15	30.00	0.17	30.00	0.03	**30.00**	**0.02**	30.00	0.03
Healthcare2	16.00	0.60	**14.00**	**0.02**	14.00	0.07	-1.00	-1.00	14.00	0.21	14.00	0.07	14.00	30.55	14.00	30.06	14.00	0.21	14.00	0.33	14.00	0.05	14.00	0.35
Healthcare3	38.00	0.58	34.00*	0.48	34.00*	0.75	34.00*	2.14	34.00*	2.39	34.00*	0.68	34.00*	1.63	34.00*	1.44	**34.00**	**0.26**	34.00	0.32	34.00	0.37	34.00	0.45
Healthcare4	49.00	0.59	46.00*	1.77	46.00*	2.33	46.00*	5.88	46.00*	7.24	46.00*	2.61	46.00*	3.64	46.00*	3.71	**46.00**	**0.73**	46.00	0.82	46.00	1.83	46.00	2.22
Insurance	**527.00**	**0.31**	527.00*	16.82	527.00*	18.56	527.00*	44.42	527.00*	45.47	527.00*	21.08	527.00*	45.23	527.00*	46.40	527.00	5.29	527.00	10.47	527.00	5.08	527.00	11.25
NetworkMgmt	112.00	0.58	110.00*	214.18	110.00*	511.52	110.00*	180.82	110.00*	233.46	110.00*	1100.79	110.00*	45.65	**110.00***	**34.19**	110.00	344.68	110.00	302.44	110.00	408.88	110.00	276.77
ProcessorComm1	29.00	0.52	21.00	67.53	22.00	0.47	-1.00	-1.00	22.00	1.14	**21.00**	**28.65**	21.00	468.17	21.00	107.68	21.00	610.20	21.00	1690.90	22.00	0.22	21.00	1015.43
ProcessorComm2	32.00	0.77	25.00*	0.42	25.00*	0.55	25.00*	1.42	25.00*	1.70	25.00*	0.42	25.00*	0.78	25.00*	1.22	**25.00**	**0.23**	25.00	0.39	25.00	0.26	25.00	0.41
Services	106.00	1.20	100.00*	28.65	100.00*	29.89	100.00*	40.55	100.00*	37.37	100.00*	26.69	100.00*	3.94	100.00*	3.97	100.00	2.43	100.00	2.44	100.00	2.42	**100.00**	**1.46**
Storage1	17.00	0.75	**17.00***	**0.00**	17.00*	0.01	17.00*	0.11	17.00*	0.11	17.00*	0.01	-1.00	-1.00	-1.00	-1.00	17.00	0.01	17.00	0.01	17.00	0.01	17.00	0.01
Storage2	18.00	0.25	**18.00***	**0.01**	18.00*	0.01	18.00*	0.12	18.00*	0.12	18.00*	0.02	18.00*	0.05	18.00*	0.05	18.00	0.02	18.00	0.01	18.00	0.01	18.00	0.01
Storage3	50.00	0.66	50.00*	0.14	50.00*	0.28	50.00*	11.02	50.00*	1.86	50.00*	0.26	-1.00	-1.00	-1.00	-1.00	50.00	0.09	50.00	0.11	**50.00**	**0.08**	50.00	0.12
Storage4	136.00	0.60	130.00*	2.49	130.00*	3.24	130.00*	7.24	130.00*	9.28	130.00*	1.69	130.00*	5.16	130.00*	6.65	**130.00**	**0.55**	130.00	0.84	130.00	0.58	130.00	0.83
Storage5	218.00	0.87	215.00	68.48	215.00	389.33	-1.00	-1.00	215.00	140.68	215.00	811.43	215.00	85.07	215.00	90.98	215.00	17.01	**215.00**	**16.83**	215.00	45.27	215.00	46.54
SystemMgmt	17.00	0.57	**15.00***	**0.01**	15.00*	0.03	15.00*	0.16	15.00*	0.18	15.00*	0.02	15.00*	0.08	15.00*	0.09	15.00	0.02	15.00	0.02	15.00	0.02	15.00	0.02
Telecom	32.00	0.58	30.00*	0.04	30.00*	0.05	30.00*	0.31	30.00*	0.37	30.00*	0.06	30.00*	0.18	30.00*	0.22	30.00	0.03	30.00	0.04	**30.00**	**0.03**	30.00	0.04
Yu et al. [102]																								
RL-A	155.00	2.88	153.00	14.02	153.00	25.95	-1.00	-1.00	153.00	28.52	153.00	18.89	153.00	58.49	153.00	63.11	**153.00**	**3.25**	153.00	3.77	153.00	12.04	153.00	15.40
RL-B	767.00	170.61	760.00	4471.65	763.00	5862.61	-1.00	-1.00	-1.00	-1.00	764.00	5633.81	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	**727.00**	**966.65**	-1.00	-1.00	727.00	5809.65
Yamada et al. [100]																								
Company2	81.00	9.37	72.00	1.80	72.00	1.88	-1.00	-1.00	72.00	3.33	72.00	2.48	72.00	32.15	72.00	32.96	**72.00**	**0.68**	72.00	0.75	72.00	0.99	72.00	1.29

TABLE 3.2: Comparison of SAT-based MaxSAT approaches versus CALOT for the Covering Array Number problem. Bold values represent the best results. In case of ties in size, the best time is marked. For sizes with a star the optimum has been certified in at least one of the five seeds executed.

	CALOT CCX a.0		CALOT CCX a.1		RC2-B CCX a.0		RC2-B CCX a.0 wpm		linear CCX a.0		loandra CCX a.0		loandra CCX a.1		tt-open-wbo-inc CCX a.0		tt-open-wbo-inc CCX a.1		tt-open-wbo-inc CCX a.0 wpm		tt-open-wbo-inc CCX a.1 wpm	
	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L	size W L	time W L
ACTS	0 **52**	2 **56**	0 **52**	2 **56**	21 **32**	23 **35**	2 **51**	4 **54**	0 **52**	2 **56**	3 **52**	4 **54**	3 **52**	4 **54**	1 **52**	2 **56**	0 **53**	1 **57**	1 **52**	2 **56**	0 **53**	1 **57**
CALOT CCX a.0	--	--	5 3	**52** 6	**21** 0	**57** 1	5 4	**53** 5	3 2	**47** 11	3 **12**	**44** 14	3 **12**	**44** 14	1 **13**	13 **45**	0 **13**	12 **46**	2 **12**	15 **43**	0 **12**	19 **39**
CALOT CCX a.1	--	--	--	--	**21** 0	**57** 1	4 3	**50** 8	5 5	27 **31**	3 **12**	42 16	3 **11**	42 16	1 **12**	9 **49**	0 **13**	6 **52**	1 **10**	7 **51**	0 **11**	11 **47**
RC2-B CCX a.0	--	--	--	--	--	--	0 **19**	31 25	0 **21**	1 **57**	2 **20**	3 **54**	2 **20**	6 **51**	0 **20**	1 **56**	0 **21**	1 **57**	0 **20**	1 **56**	0 **21**	1 **57**
RC2-B CCX a.0 wpm	--	--	--	--	--	--	--	--	3 5	7 **51**	2 **11**	8 **49**	2 **10**	10 **47**	0 **11**	1 **56**	1 **13**	4 **54**	0 9	2 **55**	1 **11**	6 **52**
linear CCX a.0	--	--	--	--	--	--	--	--	--	--	3 **11**	**41** 17	3 **11**	**41** 17	1 **12**	7 **51**	1 **13**	4 **54**	2 **11**	3 **55**	1 **12**	7 **51**
loandra CCX a.0	--	--	--	--	--	--	--	--	--	--	--	--	2 7	**38** 17	4 5	9 **48**	4 7	9 **49**	5 3	12 **45**	3 5	10 **48**
loandra CCX a.1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	5 3	13 **44**	6 5	9 **49**	7 3	13 **44**	6 5	13 **45**
tt-open-wbo-inc CCX a.0	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	2 3	**39** 19	4 2	**38** 19	2 4	**42** 16
tt-open-wbo-inc CCX a.1	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	6 3	21 **37**	2 3	**39** 19
tt-open-wbo-inc CCX a.0 wpm	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	1 4	**43** 15

TABLE 3.3: Dominance relations for CALOT and SAT-based MaxSAT approaches for the Covering Array Number problem. Bold values highlight winning algorithm per size or runtime.

a substantial improvement when using the $WPMSat^{N,t,S,lb}_{CCX}$ with the *RC2* MaxSAT solver, improving the sizes obtained by its unweighted counterpart in 19 of the 58 instances, which produces similar results than *CALOT* and $PMSat^{N,t,S,lb}_{CCX}$ *linear* approaches. This is expected since the weighted version forces RC2 to perform a top-down search as discussed in section 3.6.

We also tested the $WPMSat^{N,t,S,lb}_{CCX}$ encoding over the *tt-open-wbo-inc*, a not core-guided MaxSAT solver. We observe that results are similar or slightly worse than with the $PMSat^{N,t,S,lb}_{CCX}$. We believe the $WPMSat^{N,t,S,lb}_{CCX}$ encoding is more useful for core-guided MaxSAT solvers as it modifies their refinement strategy (i.e. improve the upper bound instead of the lower bound). We also observed that refining the lower bound for the Covering Array Number problem is more challenging than refining the upper bound, as there are some instances where encoding $PMSat^{N,t,S,lb}_{CCX}$ with *RC2* (which would refine the lower bound) is not able to report any results, usually on instances where the CAN is not found.

3.9.2 Weighted Partial MaxSAT approaches for the Tuple Number problem

Encouraged by the good results of the proposed MaxSAT approaches for the Covering Array Number problem, we now evaluate the MaxSAT approach described in section 3.7 on SAT-based MaxSAT approaches for solving the Tuple Number problem. Notice that the CALOT algorithm only works for solving the Covering Array Number problem. In this sense, this is a pioneering work on applying SAT technology to solve the Tuple Number problem.

Solvers: We choose the *tt-open-wbo-inc* MaxSAT solver to perform these experiments, as this has been the approach that achieved better results in section 3.9.1.

MaxSAT encodings: We recall there are also some variations of the $TPMSat^{N,t,S,lb}_{CCX}$ encoding, due to the way constraint CCX is formulated, i.e. the relation among c^i_τ vars and $x_{i,p,v}$ vars (see remark 1 in section 3.1). According to some preliminary experimentation we observed that variation $(c^i_\tau \leftrightarrow c^{i-1}_\tau \vee x_{i,p,v})$, to which we refer as a.2, reported also good results, while variation a.1 did not and was excluded.

We additionally noticed that, when computing the tuple number, the cost of the solution returned by the MaxSAT solver when using the original encoding of equation (a) in CCX, $(c^i_\tau \rightarrow c^{i-1}_\tau \vee x_{i,p,v})$, can indeed overestimate the real cost of the solution induced by the value of the $x_{i,p,v}$ vars, i.e., the assignments that represent the actual tests used in the solution. This can happen since it is possible to set to False a c^i_τ even if the right-hand side of the implication is True. Enforcing the other side of the implication corrects this issue. For these reasons we will use the $(c^i_\tau \leftrightarrow c^{i-1}_\tau \vee x_{i,p,v})$ variation of CCX.

Results: We would like to study the evolution of the number of covered tuples as a function of the number of tests, as we hypothesise that adding a new test close to the Covering Array Number (that guarantees all tuples can be covered) will allow adding very few additional tuples. In that sense, if these tests are expensive enough, they will not pay off in terms of the available budget and the additional percentage of coverage we can achieve.

In Figure 3.1, we show the number of tests required to reach a certain percentage of the tuples to cover for the *tt-open-wbo-inc* approach. Notice that *tt-open-wbo-inc* is an incomplete MaxSAT solver and we are therefore reporting a lower bound on the possible percentage by a particular number of tests. For lack of space, we only show the most representative instances of all the benchmark families.

We observe, for all the tested instances, that most of the tuples are covered using a relatively small number of tests and the remaining tuples require a relatively large additional number of tests. In our experiments, with only 52% of tests required for the Covering Array Number or for the best suboptimal solution from Table 3.2 in section 3.9.1, we are able to reach a 95% coverage, whereas the remaining 5% of tuples need the remaining 48% of tests.

We also notice that the Tuple Number problem is more challenging than the Covering Array Number problem. According to some experimentation that we performed using complete MaxSAT solvers, none of the tested approaches has been able to certify any optimum for $N > 1$, even for the instances that were easy to solve for the Covering Array Number problem.

Another interesting observation is the erratic behavior on the *RL-B* instance [102] (Figure 3.1, bottom right). *RL-B* is the biggest instance in the available benchmarks, having 27 parameters with domains up to 37, and with a suboptimal solution for the Covering Array Number (for $t = 2$) of 727 tests. After 100 tests, the results for the Tuple Number problem become quite unstable in contrast to the behaviour on the rest of the instances. This phenomenon points out that the approach analyzed in this section has some limitations when instances are large enough. For a fixed set of parameters, instances become bigger when we increase the strength t or the number of tests as in this case.

To conclude this section, we have confirmed that MaxSAT is a good approach to solve the Tuple Number problem with constraints. We have also observed that with a relatively small number of tests we can cover most of the tuples, and that this approach can be useful for medium-sized instances that do not need a large number of tests to reach a reasonable coverage percentage.

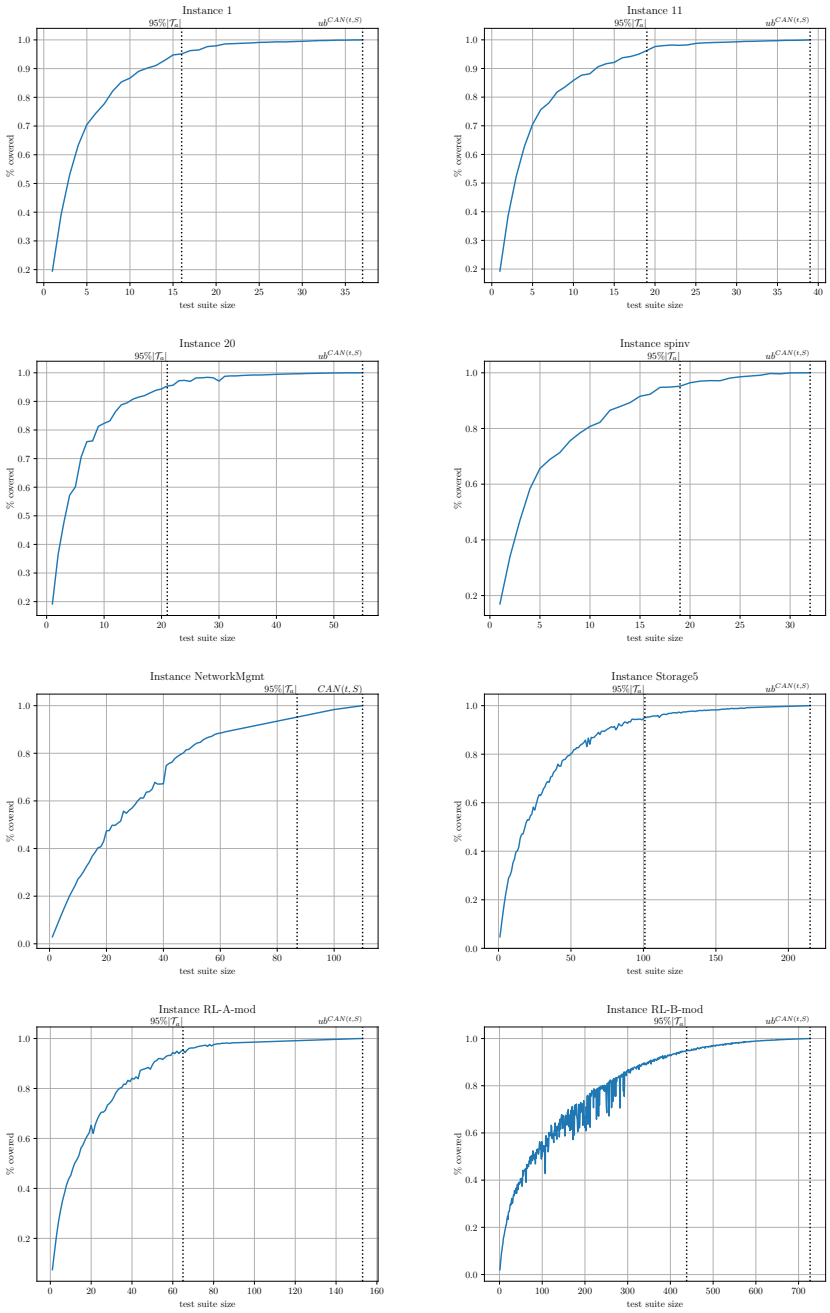
In the next section, we explore the Incremental Test Suite Construction for the Tuple Number problem described in section 3.8.1. It allows us to tackle more efficiently those Tuple Number problems involving a relatively large number of tests.

3.9.3 MaxSAT based Incremental Test Suite Construction for $T(N; t, S)$

In section 3.9.2, we have analyzed an approach that can be used to maximise the number of tuples covered by a number of tests inferior to $CAN(t, S)$. However, we have seen that it becomes less efficient if we require to compute the Tuple Number problem for a large enough number of tests.

Solving approaches: Here we propose three incomplete alternatives for solving the Tuple Number problem, with the aim of improving the results obtained in section 3.9.2. Our hypothesis is that the application of incomplete approaches can be more suitable when solving bigger instances.

FIGURE 3.1: Number of tests required to reach a certain coverage percentage for the *tt-open-wbo-inc* approach.



The first approach is the greedy algorithm presented in [100], referred to as $maxh - its$. This algorithm incrementally adds a test at a time. The test is constructed through a heuristic [41] that tries to increase the number of covered tuples so far, by selecting at each step the parameter tuple with the most value tuples yet to be covered.

The second approach is the Incremental Test Suite Construction from section 3.8.1 (referred here as $maxsat - its$), which also adds a test at a time [13], but this test is built

[13]The algorithm allows to add more than one test at a time, but this experiment is out of reach in this work.

by solving the Tuple Number problem through an incomplete MaxSAT solver instead of using a heuristic as in the previous approach.

In the third approach, instead of a MaxSAT query, as in the second approach, we apply a SAT query to return a test that covers at least one more tuple (referred to as $sat-its$) than the incremental test suite built so far.

We also evaluate the approach described in section 3.7.2. The idea is to relax the Covering Array Number problem by allowing to cover only a 95% of the allowed tuples (τ_a). We refer to this approach as $mints-95\%|\tau_a|$. As for the Covering Array Number problem, we use the upper bound returned by the ACTS tool (see section 3.2) for the initial number of tests.

Results: We present the relative performance of the previous four approaches respect to the best incomplete MaxSAT approach (*tt-open-wbo-inc*) for solving the Tuple Number problem from section 3.9.2, referred as $\simeq T(N;t,S)$ (we use the symbol $\simeq$ to indicate that the values reported for $\simeq T(N;t,S)$ correspond to suboptimal solutions). All the approaches shown in this section also use the incomplete SAT-based MaxSAT solver *tt-open-wbo-inc*, except $sat-its$ which uses the Glucose41 SAT solver. For the encoding of equation (a) of CCX we use variation a.2 ($c_\tau^i \leftrightarrow c_\tau^{i-1} \vee x_{i,p,v}$) as in section 3.9.2.

To perform a fair comparison we tried to execute all the algorithms within the same runtime conditions. We use as a reference the runtime that $maxsat-its$ needs to cover all the allowed tuples. In more detail, we set a timeout of 100s to each iteration of the $maxsat-its$ approach [14]. Therefore, the total runtime in seconds consumed by $maxsat-its$ is the number of tests it reaches multiplied by 100. For $maxh-its$ and $sat-its$, the timeout is the total runtime consumed by $maxsat-its$. For $mints-95\%|\tau_a|$, we use as timeout the runtime consumed by $\simeq T(N;t,S)$ to reach 95% of coverage. Finally, for $\simeq T(N;t,S)$, we use a timeout of $N \cdot 100$ seconds for each N. Notice that in this last case we are ensuring that for a given N, both $\simeq T(N;t,S)$ and $maxsat-its$ approaches will have the same execution time limits.

All approaches have been executed with 3 seeds and the mean is reported. The experimental results are presented in Figures 3.2 and 3.3. As in section 3.9.2, we only plot the most representative instances.

Figure 3.2 shows the increment (or decrement) of the number of tests required by $maxsat-its$, $maxh-its$ and $mints-95\%|\tau_a|$ to cover the same number of tuples as $\simeq T(N;t,S)$. On the other hand, Figure 3.3 shows the increment (or decrement) of tests required to reach the same coverage ratio as $\simeq T(N;t,S)$. For the $sat-its$ approach we found that in most cases it is able to cover only one tuple per test, so we decided to exclude these results in the figures as they were clearly outperformed by the rest of the presented approaches.

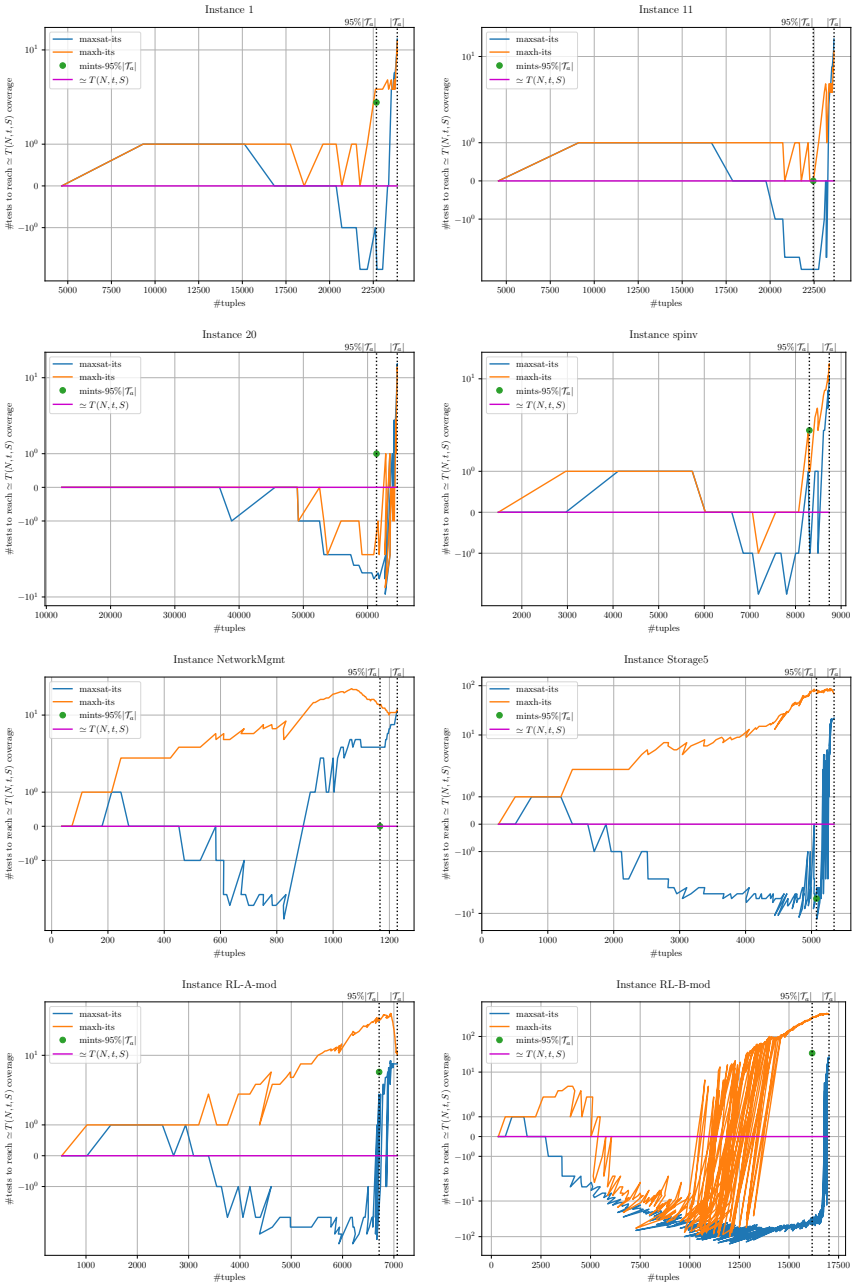
In both figures, we plot a vertical line to show the points where $\simeq T(N;t,S)$ reaches 95% and 100% of tuples covered.

In general, $maxsat-its$ clearly outperforms $maxh-its$. This can be expected since the nature of the incremental approach is to do the best at each possible iteration, and $maxsat-its$ tackles exactly this goal by solving the Tuple Number problem, while $maxh-its$ do not.

We also observe that $maxsat-its$ outperforms the tuple coverage that $\simeq T(N;t,S)$ can achieve on the first tests. Particularly, $maxsat-its$ is able to improve the number of tests required to cover 95% of the allowed tuples in 7 of the 8 instances we show in Figures 3.2 and 3.3. On the other hand, above 95%, $\simeq T(N;t,S)$ seems to be the best approach in terms of using fewer tests for the same coverage. This makes sense

[14] We assume that $maxsat-its$ is able to cover at least one more tuple in 100 seconds

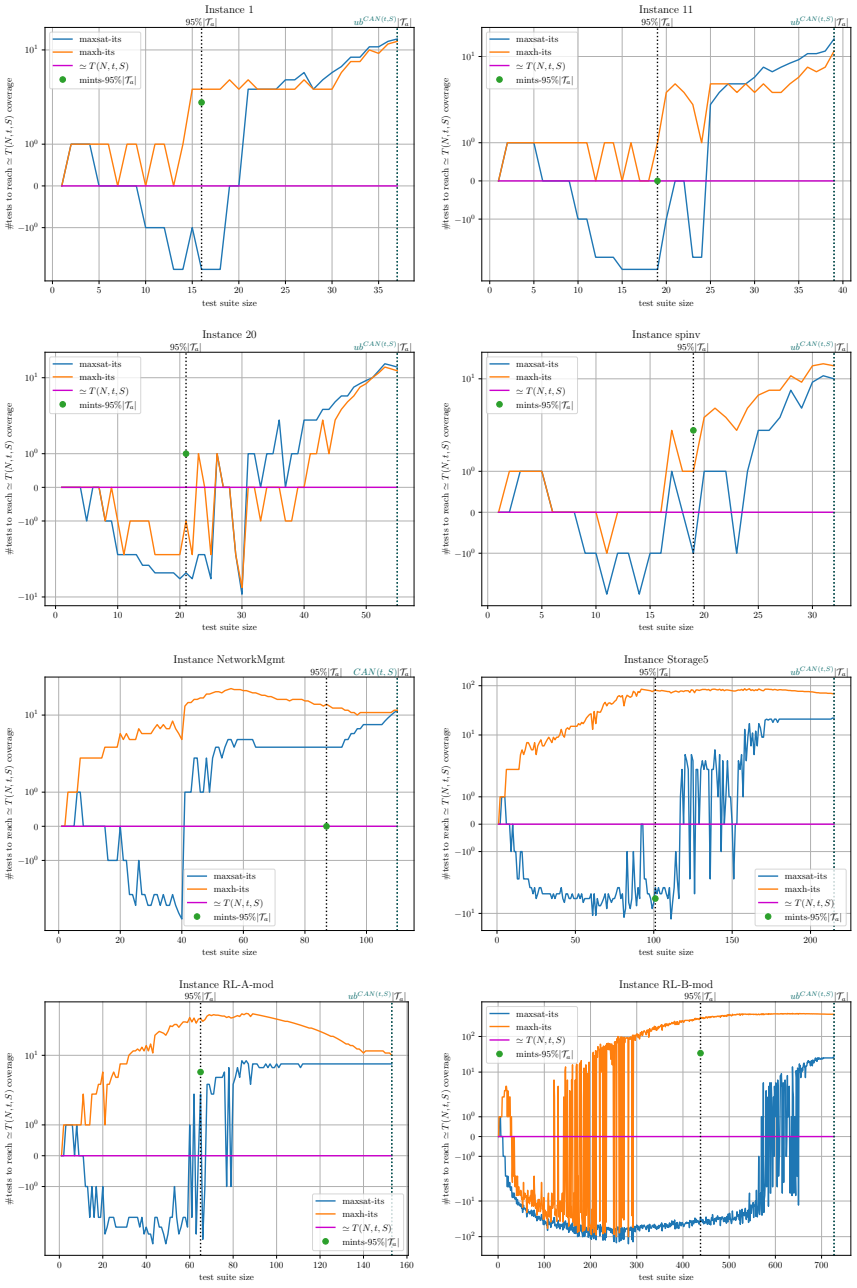
FIGURE 3.2: Comparison of the required number of tests for different methods with regards to the number of test used by $\simeq T(N,t,S)$ (as base) to cover each number of tuples.



since the incomplete nature of $maxsat-its$ makes it less efficient when approaching the complete coverage, which may not be needed for several applications.

In figure 3.2 we observe an erratic behaviour of instance *RL-B*, which is the largest instance that we had available. These results are in line with the ones in figure 3.1 of section 3.9.2, and shows the possible issues that $\simeq T(N;t,S)$ can suffer when dealing with large instances. In particular, figure 3.4 shows the number of literals of the MaxSAT instance solved by $\simeq T(N;t,S)$ and $maxsat-its$ as the size of the test suite increases for the *RL-B* benchmark. We observe that $\simeq T(N;t,S)$ has to deal with an increasing size of the Partial MaxSAT instance proportional to the number of tests in the test suite. In contrast, for $maxsat-its$ the size of the instance decreases,

FIGURE 3.3: Comparison of the required number of tests for different methods to cover as much tuples at each test from $\simeq T(N,t,S)$ (as base).



since only one test is encoded and the number of tuples to cover decreases along with the size of the test suite built so far. This is an interesting insight since RL-B instance comes from an industrial application and it may reflect what we can face in harder real-world scenarios. Therefore, $maxsat-its$ may seem more well suited for these harder real-world domains and may extend the reach of Combinatorial Testing for more complex SUTs.

Finally, although $mints-95\%|\tau_a|$ is not consistently the best option to obtain a good suboptimal test suite that covers 95% of the total tuples, it obtains the best result on instances *NetworkMgmt* and *Storage5*. Moreover, it is the only method that guarantees optimality when combined with a complete MaxSAT solver.

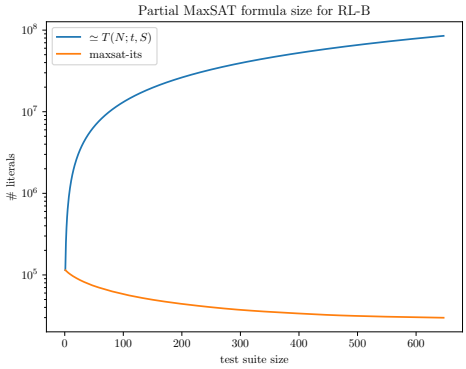


FIGURE 3.4: Partial MaxSAT formula size for RL-B in literals as a function of test suite size.

Chapter 4

Effectively Computing High Strength Mixed Covering Arrays with Constraints

In Chapter 3 we have studied how to build MCACs of minimum size for strength $t = 2$. Additionally, we have also defined a new problem called the *Tuple Number problem* that can be applied in situations where we have a limit on the number of tests that can be applied.

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 [100] (based on SAT and the domain-dependent PICT heuristic) and [15] (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.

Regarding the strength t, the size of the SAT/MaxSAT formulas in 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 $t = 5$, which clearly is a bottleneck for the mentioned SAT or MaxSAT approaches.

Finally, there are other recent Constraint Programming approaches but they focus on $t = 2$ ([59, 63]) or they do not allow SUT constraints ([69]).

In this chapter, 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 by Algorithm 5 in [100]. 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

[1] [15] can add more than one test at each iteration.

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.

Extending our work in [8], we introduce the Parallel PRBOT-its algorithm which effectively parallelizes the task of processing the different fractions of t-tuples seen by algorithm PRBOT-its. We carry out an extensive experimental evaluation that allows showing that we clearly outperform the state-of-the-art ACTS tool based on the IPOG algorithm and provide new upper bounds for some instances.

This chapter is structured as follows. Section 4.1 presents the BOT-its algorithm (Build One Test - Iterative Test Suite), an algorithm that incrementally builds MCACs test by test. Section 4.2 presents the RBOT-its algorithm that uses a MaxSAT approach to improve the BOT-its algorithm. Section 4.3 describes the PRBOT-its algorithm that shows how to adapt RBOT-its to operate on low memory requirements. Section 4.4 shows how to effectively parallelize the PRBOT-its algorithm. Finally, in Section 4.5 we study how these approaches compare to the ACTS tool.

4.1 Incremental Test Suite (ITS) Construction Algorithms

In this section we present another incomplete approach to solve the $CAN(t, S)$ problem, based on the algorithms in [100]. Unlike in the IPOG algorithm (see Section 2.3.1), these approaches iteratively build an MCAC by adding one test at a time.

Algorithm BOT-its (BOT-its: Build One Test - Iterative Test Suite), which is inspired on Algorithm 5 in [100], builds an MCAC by iteratively calling Algorithm BuildOneTest (BOT) (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 Y 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 Y

1 $Y \leftarrow \emptyset$ `# Working test suite`
2 $p \leftarrow$ pool with all t-tuples of S
3 $sat \leftarrow$ incremental SAT solver initialized with X and $SUTX$ constraints
4 **while** $p \neq \emptyset$ **do**
5 $\quad v, p \leftarrow BOT(S, p, sat, cb)$
6 $\quad Y \leftarrow Y \cup \{v\}$
7 $\quad p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$ `# Tuples in` p `covered by` v
8 $\quad p \leftarrow p \setminus p_v$
9 **return** Y

Next, we show the pseudocode for Algorithm BuildOneTest (BOT) (BOT: Build One Test), also inspired on Algorithm 5 in [100]. 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 [41] 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 [100]

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 ∪ {(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[2], 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 the 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).

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

[2]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(v)
7      while not consistent(v, ∞) do
8          (p, v) ← most recently fixed (p, v) in v s.t. (p, v) ∈ sat.core()
9          v ← v \ {(p, v)}
10     Fix unfixed parameters in v according to sat.model()
11     return v
```

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 [100], 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.

4.2 Test Suite Refinement Algorithm

In Section 4.1 we showed how Algorithm BOT-its (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 4.1), we can always try to find a smaller MCAC by using MaxSAT, as described in section 3.5. 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 (RBOT-its), which is an improvement over Algorithm 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 Y. This window also keeps track of the t-tuples ($w.p$) of the pool p covered by the window (line 11).

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

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 Y

```
1  Y ← ∅                                                  # 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      v, p ← BOT(S, p, sat, cb)
8      Y ← Y ∪ {v}
9      p_v ← {τ | τ ∈ p ∧ v ⊨ τ}                           # Tuples in p covered by v
10     p ← p \ p_v
11     w.p ← w.p ∪ p_v
12     while window_is_full(Y, w) do
13         Y, p, w ← refine(Y, p, w)
14 Y, p, w ← refine(Y, p, w)
15 return Y
```

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 Algorithm 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 3.5, 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 Y (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).

4.3 Augmenting ITS Algorithms with an 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 (PRBOT-its), an extension of Algorithm RBOT-its (see section 4.2) 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(Y, p, w)
2      φ ← encode(S, Y_{≥w.i}, w.p)
3      Y_r ← solve(φ)
4      if Y_r ≠ ∅ and |Y_r| < |Y_{≥w.i}| then
5          Replace Y_{≥w.i} by Y_r in Y
6          p_x ← {τ | τ ∈ p ∧ Y_r ⊨ τ}
7          p ← p \ p_x
8          w.p ← w.p ∪ p_x
9      else
10         v_r ← test case with index w.i in Y
11         Y ← Y \ {v_r}
12         w.p ← w.p \ {τ | τ ∈ w.p ∧ v_r ⊨ τ}
13         w.i ← w.i + 1
14     return Y, 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 4.1). 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 Algorithm BOT auxiliary functions, as follows:

$$\mathrm{Y}, p, w, \tau \leftarrow fill\_pool(\mathrm{Y}, p, w, \tau)$$

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 Y, w and output parameters w, τ.

To fill the pool of t-tuples we call function *fill_pool* in Algorithm Fill pool. This function iteratively adds new t-tuples to the pool that are neither in Y 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 Y, 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 Y

```
# All functions can access S, t and b
```
1 $Y \leftarrow \emptyset$ `# Working test suite`
2 $sat \leftarrow$ incremental SAT solver initialized with X and $SUTX$ constraints
3 $w.p \leftarrow \emptyset$ `# Window of covered tuples`
4 $w.i \leftarrow 0$ `# Window starting test index`
5 $p \leftarrow \emptyset$ `# Working pool of tuples to cover`
6 $\tau \leftarrow \emptyset$
7 **while** $p \neq \emptyset$ **or** $unseen\_tuples?(S, t, \tau)$ **do**
8 $\quad Y, p, w, \tau \leftarrow fill\_pool(Y, p, w, \tau)$
9 $\quad v, p, w, \tau \leftarrow BOT(S, p, sat, cb, Y, w)$
10 $\quad Y \leftarrow Y \cup \{v\}$
11 $\quad p_v \leftarrow \{\tau \mid \tau \in p \wedge v \models \tau\}$ `# Tuples in p covered by v`
12 $\quad p \leftarrow p \setminus p_v$
13 $\quad w.p \leftarrow w.p \cup p_v$
14 $\quad$ **while** $window\_is\_full(Y, w)$ **do**
15 $\quad\quad Y, p, w \leftarrow refine(Y, p, w)$
16 $Y, p, w \leftarrow refine(Y, p, w)$
17 **return** Y

4.4 Parallel ITS algorithms

One of the main drawbacks of the PRBOT-its algorithm is the high run-time consumption when solving large instances with high strength (i.e. $t \geq 5$). In this section, we present the last piece to get a practical application for Combinatorial Testing. In particular, we present a Parallel approach for Incremental Test Suites algorithms that leverages the power of parallel or distributed computing to build Covering Arrays in reasonable time.

Thanks to our particular design of the PRBOT-its algorithm (that operates on chunks and pools of t-tuples, see section 4.3) we can now apply a simple but effective master-worker parallel approach to scale to bigger instances and higher strengths, while slightly increasing the generated test suite size as we can see in section 4.5.

Algorithm Parallel PRBOT-its master shows the pseudocode of the master process. Essentially, we traverse all the tuples of the input SUT S and strength t (line 3). Whenever we reach the chunk size limit c (line 3) we submit a worker that will operate from the starting tuple τ_s (included) to the ending tuple τ_e (excluded, line 8)[3]. We use these *markers* since we want to avoid explicitly the set of t-tuples to the worker. The worker will generate these t-tuples thanks to the markers and an implicit order of the t-tuples (see section 4.3).

The ending tuple τ_e becomes the starting tuple for the next worker (line 9), and the tuples counter is set to 0 (line 10). Finally, we must perform a final submission with the remaining tuples (line 11).

If we are not using any queue submission engine, to avoid the concurrent submission of too many parallel jobs we use the input job limit max_w. Function `wait_for_free_slot`

[3] $\tau_s = \emptyset$ corresponds to the first tuple, and $\tau_e = \emptyset$ corresponds to the last one.

Algorithm Fill pool: Fill pool function

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

Algorithm Parallel PRBOT-its master: Master parallel PRBOT-its algorithm

```
   Input : SUT model S, strength t, chunk size c, max parallel workers max_w,
           worker args w_args
1  #tupls ← 0
2  τ_s, τ_e ← ∅, ∅
3  while unseen_tuples?(S, t, τ_s) do
4      τ_e ← next_tuple(S, t, τ_e)
5      #tupls++
6      if #tupls = c then
7          wait_for_free_slot(max_w)
8          submit(τ_s, τ_e, w_args)
9          τ_s ← τ_e
10         #tupls ← 0
11 submit(τ_s, ∅)
```

(line 7) monitors the output of the previously submitted workers, and if there are already max_w active workers it waits.

Function *submit* (line 8) submits the worker job considering the start and end tuples, as well as all the other supported arguments by the original PRBOT-its algorithm.

Regarding workers, we performed two simple modifications to the PRBOT-its algorithm explained in section 4.3. First, PRBOT-its will receive an starting and ending tuples (τ_s and τ_e respectively). Additionally, we can activate an option to load all the test suites generated by previous workers (what we called *warmstarting*). This way we can discard tuples that are already covered, which reduces substantially the size of the final test suite.

Finally, we can take even more advantage of *warmstarting* by applying what we call *progressive submission*. We start submitting just one parallel job and we wait until it finishes (i.e. enforcing a hard synchronization point). Then, we increase the number of parallel jobs by n, and we enforce *warmstarting* in all the submitted workers. Once we reach max_w jobs we deactivate all the hard synchronization points and proceed as we did without progressive submission. This way we ensure that workers load more test cases at the expense of having hard synchronization points. In section 4.5 we show how this approach helps to effectively reduce the total number of

generated tests while keeping similar or even better execution times.

4.5 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 [38], with 5 real-world and 30 artificially generated covering array problems, [96] with 20 real-world instances, [102] with two industrial instances and, [100] with another industrial instance.

In table 4.1 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). Additionally, we provide an approximation for the memory consumption of keeping all the tuples in memory for $t = 3$, $t = 4$ and $t = 5$.

We use Python as programming language and the Python framework OptiLog [7] that provides bindings to state-of-the-art SAT solvers. For our experimentation, we use Glucose 4.1. For the implementation Parallel PRBOT-its we also used Nim to implement an extension to efficiently query whether a t-tuple is already covered in the current test suite.

We implemented our own version of Algorithm BOT-its, as the implementation of Algorithm 5 described in [100] was not available from authors for reproducibility purposes[4]. 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[5]. Notice the authors used as underlying SAT solver lingeling [28] 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 [100] 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 4.1).

For the Refine function in algorithms RBOT-its and PRBOT-its we consider the encoding $PMSat^{N,t,S,lb}_{CCX}$ described in section 3.5. We use a custom implementation of the *linear* [46, 72] MaxSAT algorithm that is able to report suboptimal solutions[6], using CaDiCaL as the underlying SAT solver [30]. 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 4.2). 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 execution environment 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).

[4]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.

[5]In [100] their algorithms are implemented in C programming language

[6]Since Algorithm RBOT-its is incomplete by nature, there is actually no need to use a complete MaxSAT solver.

inst	S_P	S_φ	mem $t=3$	mem $t=4$	mem $t=5$
[38]					
1	$2^{86}3^34^15^16^2$	$2^{20}3^34^1$	20.1MB	1.4GB	74.2GB
2	$2^{86}3^34^35^16^1$	$2^{19}3^3$	15.6MB	1.0GB	48.8GB
3	$2^{27}4^2$	2^93^1	416.0kB	7.5MB	99.3MB
4	$2^{51}3^44^25^1$	$2^{15}3^2$	3.7MB	147.8MB	4.2GB
5	$2^{155}3^74^35^16^4$	$2^{32}3^64^1$	112.7MB	14.1GB	1.3TB
6	$2^{73}4^36^1$	$2^{26}3^4$	8.1MB	422.4MB	16.0GB
7	$2^{29}3^1$	$2^{13}3^2$	399.7kB	7.1MB	94.2MB
8	$2^{109}3^24^25^36^3$	$2^{32}3^44^1$	34.5MB	2.9GB	184.2GB
9	$2^{57}3^14^15^16^1$	$2^{30}3^7$	4.2MB	172.9MB	5.2GB
10	$2^{130}3^64^15^26^4$	$2^{40}3^7$	68.2MB	7.2GB	579.4GB
11	$2^{84}3^44^25^26^4$	$2^{28}3^4$	20.1MB	1.4GB	74.0GB
12	$2^{136}3^44^35^16^3$	$2^{23}3^4$	60.5MB	6.1GB	475.5GB
13	$2^{124}3^44^15^26^2$	$2^{22}3^4$	43.5MB	4.0GB	273.4GB
14	$2^{81}3^54^36^3$	$2^{13}3^2$	16.3MB	1.1GB	52.1GB
15	$2^{50}3^44^15^26^1$	$2^{20}3^2$	4.1MB	171.5MB	5.1GB
16	$2^{81}3^34^26^1$	$2^{30}3^4$	11.6MB	691.4MB	29.7GB
17	$2^{128}3^34^25^16^3$	$2^{25}3^4$	48.3MB	4.5GB	325.2GB
18	$2^{127}3^24^45^66^2$	$2^{23}3^44^1$	59.9MB	6.0GB	465.5GB
19	$2^{172}3^94^95^36^4$	$2^{38}3^5$	166.3MB	23.7GB	2.5TB
20	$2^{138}3^44^55^46^7$	$2^{42}3^6$	94.5MB	11.1GB	998.5GB
21	$2^{76}3^34^25^16^3$	$2^{40}3^6$	13MB	796.2MB	35.3GB
22	$2^{72}3^44^16^2$	$2^{20}3^2$	9.3MB	511.3MB	20.3GB
23	$2^{25}3^16^1$	$2^{13}3^2$	352.7kB	5.9MB	73.5MB
24	$2^{110}3^25^36^4$	$2^{25}3^4$	34.5MB	2.9GB	184.0GB
25	$2^{118}3^64^25^26^6$	$2^{23}3^34^1$	54.3MB	5.3GB	394.1GB
26	$2^{87}3^14^35^4$	$2^{28}3^4$	16.8MB	1.1GB	55.1GB
27	$2^{55}3^24^25^16^2$	$2^{17}3^3$	5.1MB	224.5MB	7.2GB
28	$2^{167}3^{16}4^25^36^6$	$2^{31}3^6$	160.7MB	22.7GB	2.4TB
29	$2^{134}3^75^3$	$2^{19}3^3$	52.4MB	5.1GB	374.4GB
30	$2^{73}3^34^3$	$2^{31}3^4$	8.5MB	456.4MB	17.6GB
apache	$2^{158}3^84^45^16^1$	$2^33^14^25^1$	92.5MB	10.9GB	971.3GB
bugzilla	$2^{49}3^14^2$	2^43^1	2.3MB	79.4MB	1.9GB
gcc	$2^{189}3^{10}$	$2^{37}3^3$	127.6MB	16.7GB	1.6TB
spins	$2^{13}4^5$	2^{13}	156.2kB	1.9MB	16.6MB
spinv	$2^{42}3^24^{11}$	$2^{47}3^2$	4.3MB	181.2MB	5.5GB
[96]					
Banking1	3^44^1	5^{112}	3.8kB	8.0kB	6.3kB
Banking2	$2^{14}4^1$	2^3	51.2kB	432.2kB	2.4MB
CommProtocol	$2^{10}7^1$	$2^{10}3^{10}4^{12}5^{24}$ $6^{30}7^{30}8^{12}$	26.0kB	157.5kB	616.9kB
Concurrency	2^5	$2^43^15^2$	0.9kB	1.2kB	0.6kB
Healthcare1	$2^63^25^16^1$	2^33^{18}	31.9kB	201.2kB	801.8kB
Healthcare2	$2^53^64^1$	$2^13^65^{18}$	48.2kB	379.6kB	1.9MB
Healthcare3	$2^{16}3^64^55^16^1$	2^{31}	918.8kB	21.4MB	366.0MB
Healthcare4	$2^{13}3^{12}4^65^26^17^1$	2^{22}	2.2MB	73.2MB	1.7GB
Insurance	$2^63^15^16^211^113^117^131^1$	-	1.3MB	27.3MB	363.7MB
NetworkMgmt	$2^24^15^310^211^1$	2^{20}	189.4kB	2.0MB	12.6MB
ProcessorComm1	$2^33^64^6$	2^{13}	172.7kB	2.1MB	18.5MB
ProcessorComm2	$2^33^{12}4^85^2$	1^42^{121}	1015.3kB	24.3MB	424.0MB
Services	$2^33^45^28^210^2$	$3^{386}4^2$	365.6kB	5.4MB	52.5MB
Storage1	$2^13^14^15^1$	4^{95}	1.8kB	1.9kB	0.1kB
Storage2	3^46^1	-	5.1kB	11.4kB	9.5kB
Storage3	$2^93^15^36^18^1$	$2^{38}3^{10}$	184.4kB	2.2MB	18.8MB
Storage4	$2^53^74^15^26^27^110^113^1$	2^{24}	1.0MB	23.2MB	372.2MB
Storage5	$2^53^85^36^28^19^110^211^1$	2^{151}	2.1MB	61.7MB	1.3GB
SystemMgmt	$2^53^45^1$	$2^{13}3^4$	26.7kB	162.5kB	629.8kB
Telecom	$2^53^14^25^16^1$	$2^{11}3^14^9$	43.2kB	302.9kB	1.3MB
[102]					
RL-A-mod	$2^53^44^75^46^57^48^112^3$	$1^{12}2^{491}3^{345}$	8.6MB	438.0MB	15.6GB
RL-B-mod	$2^83^24^35^36^19^1$ $10^112^214^320^124^137^1$	$1^82^{1127}3^{277}$ $4^{1755}5^{1064}6^{2048}$	16.4MB	956.6MB	37.9GB
[100]					
Company2	$2^63^48^4$	$1^22^{35}3^{89}4^{54}5^{34}$ $6^{20}7^{34}8^{16}9^4$	247.9kB	3.2MB	28.4MB

TABLE 4.1: SUT parameters domains and constraints for each instance (columns S_P and S_φ) and memory consumption for $t=3$, $t=4$ and $t=5$ (*mem*).

4.5.1 PRBOT-its evaluation

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 with its default configuration. We describe the results in Table 4.2 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 Algorithm 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 required by other auxiliary data structures or by the SAT solver called within BOT-its. Table 4.1 shows 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 Algorithm 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[7].

Overall, we found that ACTS reports MCACs in 49 more instances than RBOT-its and PRBOT-its. However, we may be observing a 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 of 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*

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

	$t = 3$								
	ACTS			BOT-its			RBOT-its		
inst	tests	%	time	tests	%	time	tests	%	time
[38]									
1	**293**	**100%**	**4s**	294.20	100%	12m	294.20	100%	1.3h
2	174	100%	3s	176.50	100%	6m	**149.10**	**100%**	**39m**
3	71	100%	1s	72.90	100%	4s	**50.50**	**100%**	**5m**
4	102	100%	2s	108.10	100%	48s	**81.10**	**100%**	**7m**
5	386	100%	14s	**384**	**100%**	**1.6h**	384	100%	3.3h
6	119	100%	2s	133.20	100%	2m	**98.60**	**100%**	**14m**
7	35	100%	1s	39	100%	3s	**28.40**	**100%**	**3m**
8	326	100%	5s	306.60	100%	23m	**306.20**	**100%**	**1.1h**
9	84	100%	2s	94.30	100%	44s	**60**	**100%**	**4m**
10	**329**	**100%**	**9s**	342.60	100%	51m	341.30	100%	2.4h
11	**318**	**100%**	**4s**	328.70	100%	13m	328.60	100%	1.4h
12	263	100%	7s	269.80	100%	36m	**250**	**100%**	**1.6h**
13	200	100%	7s	214.40	100%	19m	**183.70**	**100%**	**1.0h**
14	244	100%	3s	244.30	100%	7m	**216.30**	**100%**	**20m**
15	173	100%	2s	180.10	100%	1m	**150.90**	**100%**	**5m**
16	117	100%	3s	138.50	100%	3m	**96.40**	**100%**	**9m**
17	265	100%	6s	263.50	100%	30m	**239.40**	**100%**	**1.3h**
18	344	100%	8s	**327.20**	**100%**	**41m**	327.20	100%	2.1h
19	373	100%	21s	385	100%	2.6h	**365.50**	**100%**	**6.7h**
20	**463**	**100%**	**12s**	465.60	100%	1.5h	465.60	100%	4.3h
21	235	100%	3s	235.40	100%	5m	**216.50**	**100%**	**17m**
22	164	100%	2s	164.70	100%	3m	**144**	**100%**	**8m**
23	48	100%	1s	55.40	100%	3s	**37.30**	**100%**	**3m**
24	341	100%	5s	**337.70**	**100%**	**25m**	337.70	100%	1.6h
25	**404**	**100%**	**7s**	407.70	100%	47m	407.70	100%	2.6h
26	207	100%	3s	205.10	100%	7m	**195.30**	**100%**	**47m**
27	204	100%	2s	210.90	100%	2m	**180.50**	**100%**	**10m**
28	**420**	**100%**	**21s**	421.80	100%	2.6h	421.80	100%	4.6h
29	154	100%	5s	156.10	100%	20m	**125.70**	**100%**	**43m**
30	100	100%	2s	93.70	100%	2m	**73.80**	**100%**	**14m**
apache	173	100%	9s	191.60	100%	36m	**168.20**	**100%**	**1.7h**
bugzilla	68	100%	1s	72.20	100%	22s	**49.50**	**100%**	**9m**
gcc	108	100%	10s	121	100%	43m	**81.80**	**100%**	**1.4h**
spins	**98**	**100%**	**1s**	112.80	100%	2s	105.60	100%	3m
spinv	286	100%	2s	251.70	100%	2m	**238.90**	**100%**	**1.2h**
[96]									
Banking1	58	100%	2s	55.10	100%	0s	**45**	**100%**	**30s**
Banking2	39	100%	1s	44.70	100%	0s	**30**	**100%**	**3m**
CommProtocol	49	100%	3s	50.30	100%	0s	**41**	**100%**	**3m**
Concurrency	**8**	**100%**	**1s**	**8**	**100%**	**0s**	**8**	**100%**	**0s**
Healthcare1	105	100%	1s	107.50	100%	0s	**96**	**100%**	**9s**
Healthcare2	67	100%	1s	68.40	100%	0s	**54.80**	**100%**	**3m**
Healthcare3	209	100%	1s	205.70	100%	15s	**177.10**	**100%**	**41m**
Healthcare4	294	100%	1s	309	100%	39s	**274.90**	**100%**	**53m**
Insurance	6866	100%	1s	6861.10	100%	3m	**6858.40**	**100%**	**15m**
NetworkMgmt	1125	100%	1s	1107.70	100%	4s	**1100.40**	**100%**	**2m**
ProcessorComm1	163	100%	1s	144.10	100%	2s	**131.60**	**100%**	**3m**
ProcessorComm2	161	100%	2s	169.30	100%	11s	**145.50**	**100%**	**31m**
Services	963	100%	6s	**926.80**	**100%**	**13s**	926.80	100%	5.7h
Storage1	**25**	**100%**	**2s**	**25**	**100%**	**0s**	**25**	**100%**	**0s**
Storage2	74	100%	0s	71.50	100%	0s	**54**	**100%**	**1s**
Storage3	239	100%	1s	239.20	100%	3s	**222**	**100%**	**9m**
Storage4	990	100%	1s	970.40	100%	28s	**916.40**	**100%**	**15m**
Storage5	**1879**	**100%**	**4s**	1936.10	100%	3m	1000.50	96%	12h
SystemMgmt	60	100%	1s	58.10	100%	0s	**45**	**100%**	**2s**
Telecom	126	100%	1s	125.20	100%	0s	**120**	**100%**	**5s**
[102]									
RL-A-mod	1132	100%	16s	1079.40	100%	4m	**1069.20**	**100%**	**7.8h**
RL-B-mod	14977	100%	4m	**13319.40**	**100%**	**3.1h**	4954	92%	12h
[100]									
Company2	**424**	**100%**	**15s**	432.50	100%	7s	427.20	100%	54m

TABLE 4.2: 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.

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

TABLE 4.3: 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.

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 $t = 3$.

4.5.2 Parallel PRBOT-its evaluation

As we can observe in Table 4.3, there are 18 instances for strength $t = 5$ where none of the tested methods is able to complete an MCAC. For the final part of our experimental evaluation, we will try to find MCACs for these instances by extending the memory and time limits described in section 4.5.

In particular, we extended the memory and run-time limits to 1TB and 48h, and changed the environment of execution to a cluster with nodes equipped with two AMD 7403 processors (24 cores at 2.8 GHz) and 21 GB of RAM per core.

We used the parallel PRBOT-its algorithm described in section 4.4, and we compared against the results of ACTS with extended resources. For the parallel PRBOT-its algorithm, we set a memory limit of 20GB per worker and used 48 workers plus one master job[8].

We tested three different versions of the parallel PRBOT-its algorithm to assess the effectiveness of the proposed improvements in section 4.4. The first one only applies *warmstarting* to the PBOT-its algorithm (*P-PBOT-its (ws)*), the second one adds *progressive submission* (*P-PBOT-its (ws + prog)*) and the last one adds MaxSAT refinement (*P-PRBOT-its (ws + prog)*). For all the approaches we set a pool limit of 1GB, and each worker operates on a chunk size limit c of 10 pools. For the progressive submission, we increment one job after each synchronization until we reach the maximum number of parallel jobs. For the refinement approach, we set a window limit of 500MB and only apply a refinement of 300s at the end of each worker.

We noticed that instance *RL-B* needed more memory resources per worker due to the size of the generated MCAC, so we reduced the number of parallel workers from 48 to 24 and extended their memory limit from 20GB to 40GB. We also reduced the chunk size limit c per worker from 10 to 2 and extended the total time limit from 48h to 72h.

Table 4.4 shows the results of our experimentation. We noticed that ACTS was able to report MCACs only in 2 of the 18 selected instances for $t = 5$, whereas Parallel PRBOT-its obtains MCACs in all of them. Particularly, ACTS timeouts in 7 of the 16 unsolved instances, while for the 9 remaining it would need more than 1TB of memory.

Regarding MCAC sizes, ACTS obtains better results in the two instances that it is able to solve. For the rest, *P-PRBOT-its (ws + prog)* clearly outperforms the two variants of *P-PBOT-its* without refinement, being able to report an MCAC on instances where both *P-PBOT-its (ws)* and *P-PBOT-its (ws + prog)* timeout (instances 19, 28 and RL-B) and greatly improving the sizes on the rest.

For the run-times, we observe how all the versions of the parallel PRBOT-algorithm obtain better times than ACTS. This is an interesting result, as Parallel PRBOT-its can be used to generate MCACs quicker than ACTS, at the expense of an increase on their sizes.

[8]With this setup, we are using the same resources as a single ACTS job consuming 1TB of memory

	ACTS		P-PBOT-its (ws)		P-PBOT-its (ws + prog)		P-PRBOT-its (ws + prog)	
inst	tests	time	tests	time	tests	time	tests	time
[38]								
1	**8592**	**17h**	378306	2h	157562	3h	57652	4h
5	M OUT	-	1465455	40h	1460093	41h	**258196**	**19h**
8	M OUT	-	517154	3h	270236	4h	**83429**	**6h**
10	M OUT	-	831583	11h	817998	13h	**153379**	**9h**
11	**10252**	**22h**	402037	2h	180094	4h	56464	5h
12	M OUT	-	757465	8h	726851	10h	**83472**	**8h**
13	M OUT	-	368530	3h	326664	4h	**40171**	**6h**
17	M OUT	-	458383	5h	434318	6h	**68608**	**8h**
18	T OUT	48h	997527	7h	949027	9h	**126586**	**8h**
19	T OUT	48h	T OUT	48h	T OUT	48h	**381470**	**46h**
20	T OUT	48h	1573128	21h	1513108	22h	**401293**	**16h**
24	T OUT	48h	557110	3h	368689	4h	**96652**	**6h**
25	T OUT	48h	1121699	8h	1072473	10h	**230558**	**9h**
28	T OUT	48h	T OUT	48h	T OUT	48h	**449609**	**38h**
29	M OUT	-	522676	6h	485201	8h	**44068**	**6h**
apache	M OUT	-	1155093	23h	1089386	25h	**60391**	**13h**
gcc	M OUT	-	946247	39h	889371	45h	**50241**	**21h**
[102]								
RL-B	T OUT	72h	16163442	57h	15085848	67h	**14736438**	**72h**

TABLE 4.4: Results for the unfinished instances in Table 4.3 for $t = 5$ with extended limits. MCAC size and run-time are reported for each approach. Best results are marked in bold.

Another interesting result is the decrease on run-time of the *P-PRBOT-its (ws + prog)*. Even though this approach has a call to a MaxSAT solver of 300s on each worker, it is able to reduce the run-times with respect to the other *P-PBOT-its* variations on many instances. This result shows that an early improvement on the sizes of the intermediate test suites can also be positively reflected on the run-times.

To the best of our knowledge, this is the first work where MCACs are obtained for $t = 5$ in the 16 instances in Table 4.4 where ACTS was not able to find a solution. We will make available to the community the smallest MCACs that we found for these instances.

Chapter 5

A Benchmark Generator for Combinatorial Testing

One of the main drawbacks that we found when designing Combinatorial Testing (CT) tools, which are used to test other tools, is the absence of a rich collection of SUT benchmarks to test empirically the correctness and goodness of these CT tools. This is particularly true when we deal with CT tools for SUTs with constraints since unless we obtain benchmarks (SUTs with constraints) from the industry field, we have to *artificially* generate them. Moreover, we also want to control the hardness of the constraints associated with the SUT.

In the literature, we find a limited collection of benchmarks for SUT with constraints. These benchmarks consist of 28, relatively easy, real-world benchmarks [38, 96, 102, 100] and 30 artificial benchmarks that were artificially generated considering the features of some of the first ones [38].

The absence of a rich collection of benchmarks, in contrast to other disciplines, such as in the Satisfiability research community [33], is a source of an endless list of disadvantages. Prevents the CT community from having a deeper understanding of the competitiveness of CT tools and slows down the development of new algorithms and tools. Moreover, current CT tools can be biased toward solving problems that are similar to the reduced set of available benchmarks, and, last but not least, makes it difficult to run competitions in the field, events that historically have boosted the research in other disciplines[1].

In this paper, we present a new generator for SUTs with constraints. We take an original approach since the SUT will actually be a *subproblem* of an existing decision combinatorial problem. In particular, we will focus on the SAT problem and SAT instances. In this sense, we fix some variables in the SAT instance and simplify the formula by applying some incomplete inference Boolean mechanism such as Unit Propagation. From the variables in the new remaining subformula, we select some Boolean variables to be the input parameters of the new SUT and the rest to be auxiliary variables. The clauses in the subformula become the constraints of the new SUT.

Our approach allows in a natural way access to the rich diversity of constraints that belong to the plethora of industrial, crafted and random SAT instances available in the SAT community.

Moreover, the current SUT benchmarks available in the literature do contain a set of constraints that we can consider *easy* from the point of view of its computational hardness. We should expect these sets of constraints to be *harder* as CT techniques

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

are applied to more complex real-world scenarios. In this sense, we show how to control empirically the hardness of the constraints included in the SUT.

Equipped with this new generator of SUTs with constraints, we generate a new set of SUT benchmarks and conduct an extensive evaluation on some of the available CT tools. In particular, we focus on the evaluation of the IPOG [73] and BOT-its [8, 100] algorithms. Thanks to our experimental investigation we are able to come up with some new interesting recipes on when to use a particular CT tool to test a given SUT. As an additional contribution, we will make available to the community the generated benchmarks as well as the generator.

This paper is structured as follows: Section 5.1 presents our approach to generate SUT with constraints benchmarks. First of all, we review some of the most widely-used formats to represent SUT with constraints and extend one of them in Section 5.1.1 to make more convenient the representation of the generated SUT constraints. Then, in Section 5.1.2 we describe in detail the *SUT-G* generator. Later, in our experimental evaluation (Section 5.2.2), we use the IPOG [73] and BOT-its [7, 100] MCAC algorithms to solve benchmarks generated with the *SUT-G* generator and try to provide another point of view on the their behaviour. Thanks to this experimental evaluation, we are able to provide in Section 5.3 several recipes to effectively apply MCAC generation tools to real-world scenarios.

5.1 Generating SUTs with Constraints

In this section, we present the *SUT-G* generator, a novel benchmark generator that can craft Systems Under Test (SUT) models with constraints from any SAT instance. First of all, we review some of the current state-of-the-art formats to represent SUTs with constraints and propose an extended format to represent SUT constraints more conveniently in some scenarios. Finally, we describe the *SUT-G* generator, the second main contribution of this paper.

5.1.1 Available formats for representing SUTs

We review four of the most widely used formats for representing SUT models.

The CASA format

CASA [55] is CT tool for building Covering Arrays through Simulated Annealing. This tool defines a SUT format[2] that is still used by some recent tools such as WCA [48] or AutoCCAG [81].

This format consists on two separate files: the `.model` file, that defines the SUT parameters and their domains, and the `.constraints`, which define the SUT constraints.

For the `.model` file, we must define the number of parameters of the SUT, the desired *strength* of the interactions, and the domain of the parameters. No more information can be added to the `.model` file. In Figure 5.1 we show the associated CASA model file for Example 5 and strength $t = 2$.

Regarding constraints, these can only be represented as conjunctions of disjunctions (i.e. CNF format). We must specify the total number of clauses, and for each

[2]The full CASA format specification can be found in http://cse.unl.edu/~citportal/citportal/academic

```
4
2
5 4 4 2
```

FIGURE 5.1: CASA model file for Example 5

clause its number of symbols. These symbols (also named as literals) can only refer to parameters' values, and they are derived from the parameter order defined in the model file, starting from 0. Literals can be positive or negative. For example, in Example 5, $OS = Linux$ would be symbol 1, $OS = Windows$ symbol 2, and so on. Figure 5.2 shows the CASA constraints for Example 5, where we have manually converted the SUT constraints to CNF.

```
9
2
- 0 + 14
2
- 1 + 14
2
- 2 + 14
2
- 0 - 8
2
- 1 - 8
2
- 2 - 8
2
- 4 - 9
2
- 3 - 9
3
- 6 + 2 + 3
```

FIGURE 5.2: CASA constraints file for Example 5

Notice that this format for SUT constraints does not allow auxiliary variables (i.e. all the literals that appear in the constraints must be part of the defined parameters). As we will discuss in Section 5.1.1, auxiliary variables are very useful to represent certain SUT constraints in a more convenient way.

The PICT format

PICT [41] is another CT tool that also defines its own format for representing SUT models[3].

This is a more advanced format than CASA and offers more features in terms of the definition of the SUT.

In this case, all the SUT is defined using a single `.pict` file. Unlike in CASA, in PICT we can define the name of the parameter, its type (*string* or *numeric*) and, for *strings*, the name of the parameter's values. This can be useful to better understand the definition of the SUT, as well as the output test suite.

[3]The full specification for the PICT format can be found here: https://github.com/microsoft/pict/blob/main/doc/pict.md

```
OS: L, W, M, i, A
Pl: F, S, C, A
Re: K, F, H, W
Or: P, L

IF ([OS] = "L" OR [OS] = "W" OR [OS] = "M")
THEN ([Or] = "L" AND [Pl] <> "A")
IF [Pl] = "S" THEN ([OS] = "M" OR [OS] = "i")
IF ([OS] = "i" OR [OS] = "A") THEN [Re] <> "K"
```

FIGURE 5.3: Representation of SUT in Example 5 in PICT format

Regarding SUT constraints, PICT offers *if-then* operands, as well as negations, conjunctions and disjunctions. SUT constraints are the conjunction of all the defined constraints.

PICT also provide additional features for its format such as negative testing or weighting. Its full definition can be found here: https://github.com/microsoft/pict/blob/main/doc/pict.md

Figure 5.3 shows Example 5 expressed in PICT format.

The CTWedge format

CTWedge [53] is a CT tool that also defines its own SUT format[4].

As in PICT, CTWedge also allows to name parameters and values, as well as to define their type (which can be *Bool*, *Enumerative* and *Range*).

To define constraints, it also allows to use negation, conjunction, disjunction and implication operands, and SUT constraints are the conjunction of all the defined constraints. As in all the previous cases, this format also does not support auxiliary variables.

Figure 5.4 shows Example 5 represented using the CTWedge format.

```
Model MySUT

OS: { L W M i A }
Pl: { F S C A }
Re: { K F H W }
Or: { P L }

Constraints:
    # (OS == "L" || OS == "W" || OS == "M") =>
        (Or == "L" && Pl != "A") #
    # Pl == "S" => (OS == "M" || OS== "i") #
    # (OS == "i" || OS == "A") => Re != "K" #
```

FIGURE 5.4: Representation of SUT in Example 5 in CTWedge format

[4]The grammar of the CTWedge format can be found here: https://github.com/fmselab/ctwedge/blob/master/ctwedge.parent/ctwedge/src/ctwedge/CTWEdge.xtext

```
[System]
Name: MySUT
[Parameter]
OS (enum) : L,W,M,i,A
Pl (enum) : F,S,C,A
Re (enum) : K,F,H,W
Or (enum) : P,L
[Constraint]
C1: (OS == "L" || OS == "W" || OS == "M") =>
    (Or == "L" && Pl != "A")
C2: Pl == "S" => (OS == "M" || OS== "i")
C3: (OS == "i" || OS == "A") => Re != "K"
```

FIGURE 5.5: Representation of SUT in Example 5 in ACTS format

The ACTS format

ACTS [34] is one of the most widely used CT tools for building Covering Arrays. It implements several CT algorithms, and also defines its own format for defining SUTs[5].

As in PICT and CTWedge, in ACTS we can define the name of the parameter, its type (int, enum or bool) and, for *enums*, the name of the parameter's values.

Regarding SUT constraints, ACTS allows the usage of implications, conjunctions and disjunctions to define a constraint. SUT constraints are defined as the conjunction of each of the defined constraints. All the symbols that appear in the SUT constraints must refer to some defined parameter and value, and as we have seen in all the previous formats, auxiliary variables cannot be represented using ACTS.

The ACTS format also has additional features, such as support for mixed interaction strength (where the user can specify different strengths for different subsets of parameters).

In Figure 5.5 we show Example 5 represented using the ACTS format.

The new *Extended* ACTS format

Here, we propose an extension of the ACTS format to describe SUTs with constraints.

As we have seen in Section 5.1.1, none of the reviewed formats supports the addition of *auxiliary variables* to the SUT constraints. In other words, all the symbols that appear in the SUT constraints must refer to some parameter's value in the SUT. This can be an important limitation of the current formats, as it will be more convenient to represent certain constraints using auxiliary variables.

Essentially, we propose a simple extension to the ACTS format described in Section 5.1.1. In particular, we define a new section on the format under the tag `[Auxiliar]` where the user can define the list of auxiliary variables that will appear in the set of constraints. Auxiliary variables are defined using the same syntax as parameters and can be of any of their types. Then, both the parameters and the auxiliary variables can be freely used in the SUT constraints.

In the next section we present the *SUT-G* generator, which uses the *Extended ACTS* format to represent the generated SUT constraints.

[5]The full specification of the ACTS format can be found in https://csrc.nist.rip/groups/SNS/acts/documents/acts_user_guide_2.92.pdf

5.1.2 The *SUT-G* Generator

The *SUT-G* generator that we present in this section generates a SUT instance taking as input a SAT instance.

Algorithm SUT-G: SUT Generation algorithm from SAT instances

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

Algorithm SUT-G shows the pseudocode of the proposed *SUT-G* generator. Essentially, this generator receives a SAT formula φ and returns a SUT model with constraints, where this input formula φ has been adapted to the desired difficulty. We need a *hardness measure* to adapt φ, and in our case, we decided to use the number of conflicts the SAT solver needs in average to solve the instance.

Aside from φ, *SUT-G* receives as input parameters the number of parameters n of the output SUT, the maximum and minimum number of conflicts of the SUT constraints (c_{max} and c_{min}), and other parameters that control several aspects of SUT generation that will be explained later ($\Delta_{\mathcal{A}}$, $\nabla_{\mathcal{A}}$, $\mathcal{S}$ and MAX_TRIES).

On a high level, the *SUT-G* generator will try to find a satisfiable subproblem over the input formula φ (line 1). This subproblem will match all the requirements specified in the input parameters of the algorithm. In case the input formula is unsatisfiable or if the requirements cannot be fulfilled, the `find_satisfiable_subproblem` method will return a *fail* status and the generator will exit. Otherwise, `find_satisfiable_subproblem` will return a *success* status and a list of assumptions that will adapt the input formula to the hardness requirements when applied to the original formula φ.

To apply this modifications to the original formula, we just add as unit clauses each literal in $\mathcal{A}$ (lines 4 and 5). Then, we *simplify* this formula to obtain φ' (line 6), which will become the SUT constraints.

This simplification process propagates all the unit clauses in the formula and obtains a list of literals *lits* with a fixed value. Then, it iterates all the clauses in φ and eliminates all the ones that have a literal in *lits* (i.e. this clause is satisfied and can be ignored in the resulting formula). On the other hand, if a literal in a clause appears with opposite polarity in *lits* we can remove this literal from the clause.

Finally, we rename the variables in the new formula to ensure that all the variables from 1 to $\varphi'.n\_vars$ appear in the constraints.

To finish the generation of the SUT, we just have to select its parameters. In this case they are randomly selected from 1 to $\varphi'.n\_vars$ taking into account the number of parameters n specified by the user (line 7). The rest of the variables in φ' will be auxiliary variables.

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

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|⟩
```

Function `find_satisfiable_subproblem` (used in line 1 of Algorithm SUT-G) starts initializing an incremental SAT solver with the input formula φ, as well as the set of assumptions $\mathcal{A}$ to the empty set (lines 2 and 3). Then, it uses the function `solve_subproblem` to find the *hardness* of the subproblem in terms of conflicts c, which is defined by the original formula φ and the set of assumptions $\mathcal{A}$[6] (line 4). Additionally, `solve_subproblem` will return a model that will be used later.

In case the original formula is UNSAT (which we represent when the *model* is empty), or there are not enough *unfixed* variables that could become parameters of

[6]Initially $\mathcal{A}$ is empty, this is equivalent to solving the original formula φ

the SUT[7] ($sat.n\_vars - |sat.propagate(\mathcal{A})| < n$), or the formula is too easy ($c < c_{min}$), the function will return a *fail* status (line 5).

Lines 6 - 15 show the main loop of the function. We consider a maximum number of tries MAX_TRIES to obtain the desired subproblem (which is a parameter of *SUT-G*, see Algorithm SUT-G). In case these MAX_TRIES are reached and the desired subproblem could not be obtained, we also return a *fail* status, as shown in line 16).

At each iteration of the main loop, the algorithm tries to find a subproblem by increasing or decreasing the assumptions $\mathcal{A}$ that will contain enough variables (to become the parameters in the SUT) and with the desired hardness in terms of the number of conflicts.

The idea is that by increasing the assumptions we are making the problem *easier*, and decreasing them *harder*. This is however a greedy approach, and there might be the case where the problem becomes *harder* when increasing the constraints, and vice-versa. Nonetheless, we found this greedy process quite successfully when generating interesting SUTs, as we show in our experimental evaluation (see Section 5.2.2).

In line 8 we check if there are not enough *unfixed* variables that could become parameters of the SUT ($sat.n\_vars - |sat.propagate(\mathcal{A})| < n$), or the current subproblem is too easy ($c < c_{min}$). If that is the case, we try to decrease the assumptions $\mathcal{A}$. In case we have some literals in the assumptions, we remove $\nabla_{\mathcal{A}}$ random literals in $\mathcal{A}$ in line 10, which is one of the input parameters of *SUT-G* (see Algorithm SUT-G).

However, it might be the case where the assumptions that we have at this point are empty (line 9). This means that we cannot simplify the subproblem anymore, and therefore we return a *fail*. Notice that this case is possible due to the incremental nature of the SAT solver. At first, we might find that the original formula is too hard in line 4, so in the main loop, we will increase the assumptions (as it will be explained next). Then, the SAT solver might *learn* some clauses that make easier the original problem[8]. At this point, decreasing assumptions will not help to increase the hardness of the subproblem, so when we reach $|\mathcal{A}| = 0$ we can stop the loop.

On the other hand, if the current subproblem is too hard ($c > c_{max}$, line 11), we increase the assumptions $\mathcal{A}$ by adding to them $\Delta_{\mathcal{A}}$ random literals of the *model* returned by `solve_subproblem`. ($\Delta_{\mathcal{A}}$ is another input parameter of *SUT-G*, see Algorithm SUT-G).

If none of these cases is found, we are within the conflicts range and there are enough *unfixed* variables that can become parameters of the output SUT. Therefore, we just return a *success* state and the list of *propagated* assumptions (line 12).

Finally, in lines 14 and 15 we call `solve_subproblem` again to update the number of conflicts c and the model according to the changes in $\mathcal{A}$ that we performed, and we add one try to *tries*. Notice however that these two lines are only executed in case there are enough *unfixed* variables in the subproblem (line 13). If this is not the case, we will keep decreasing $\mathcal{A}$ without solving the subproblem nor increasing the number of tries.

In line 18 we describe function `solve_subproblem`. This function will solve each subproblem and estimate its hardness in terms of conflicts.

First, it starts initializing the number of conflicts c and a set of models *models* to 0 and $\emptyset$ respectively (lines 19 and 20).

Then, in line 21 it iterates $\mathcal{S}$, which contains a list of seeds provided by the user as input in *SUT-G*. We set the Random Number Generator of the SAT solver to the

[7] We are also considering as *fixed* variables that cannot become SUT parameters the Unit Propagation of $\mathcal{A}$. If some of these variables is selected as a parameter of the SUT it will have just one possible value.

[8] These clauses are kept between calls to *solve*

current *seed* in line 22, and proceed to solve the current subproblem in line 23 (which is the formula in the SAT solver plus $\mathcal{A}$). In this same line, we check if the subproblem is UNSAT, and return an empty model if this is the case. Otherwise, when the subproblem is SAT, we retrieve the model and add it to *models* in line 24. Additionally, we compute the number of conflicts and sum them to c in line 25. At the end in line 26 we return one random model from the list of models, as well as the average on the number of conflicts [9].

For this version of the algorithm, we are considering that all the parameters in the SUT have domain 2. In next versions, we will consider the generation of SUT models with mixed domains.

We present an example of a generated benchmark using the *SUT-G* generator.

Let the DIMACS[10] formula shown in Figure 5.6 be the input SAT formula φ to *SUT-G*. This corresponds to the *AProVE09-21* SAT instance, extracted from the SAT2009 competition [95], which describes a Termination Analysis problem for Term Rewrite Systems (TRSs) [50].

```
p cnf 29964 91044
29964 0
29964 -29842 -29963 0
-29964 29842 0
...
29962 -11821 0
-29961 -11763 0
29961 11763 0
```

FIGURE 5.6: Example input formula φ

As we can observe from the header of the file (`p cnf 29964 91044`), this input formula has 29964 variables and 91044 clauses.

To exemplify the usage of *SUT-G*, we will generate the *AProVE09-21* SUT benchmarks that appears in Table 5.7 of Section 5.2, wich has 100 SUT parameters. Figure 5.7 shows a portion of this generated benchmark[11].

The *SUT-G* generator will output the SUT instance in the *Extended ACTS* format (see Section 5.1.1), as it is more convenient to represent this kind of constraints using auxiliary variables. We observe a reduction in the number of variables and clauses with respect to φ (from 29964 variables to 28301, and from 91044 clauses to 85144). This is the result of finding a satisfiable subproblem that meets the requirements in terms of the required hardness window.

In this case, we have generated a SUT with 100 binary parameters (declared in the `[Parameter]` section of the SUT). The rest of the variables have been declared as *auxiliary* variables in the `[Auxiliar]` section of the SUT. Then, under the section `[Constraint]`, we have specified the SUT constraints, both referring to *parameters* and *auxiliary* variables.

[9] We return the average on the number of conflicts over all the $\mathcal{S}$ to mitigate lucky and unlucky runs of the SAT solver

[10] `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps`

[11] We will make available to the community all the generated benchmarks, as well as the *SUT-G* generator

```
[System]
Name: AProVE09-21
[Parameter]
p70 (bool): TRUE,FALSE
p302 (bool): TRUE,FALSE
p705 (bool): TRUE,FALSE
...
p27729 (bool): TRUE,FALSE
p27961 (bool): TRUE,FALSE
p28179 (bool): TRUE,FALSE
[Auxiliary]
p1 (bool): TRUE,FALSE
p2 (bool): TRUE,FALSE
p3 (bool): TRUE,FALSE
...
p28299 (bool): TRUE,FALSE
p28300 (bool): TRUE,FALSE
p28301 (bool): TRUE,FALSE
[Constraint]
C1: p1 == TRUE || p2 == TRUE
C2: p1 == TRUE || p3 == FALSE || p4 == FALSE
C3: p1 == FALSE || p3 == TRUE
...
C85142: p28301 == TRUE || p11219 == TRUE
C85143: p28301 == FALSE || p11158 == FALSE
C85144: p28301 == TRUE || p11158 == TRUE
```

FIGURE 5.7: Example output SUT for *SUT-G*

5.2 Assessing MCAC Tools with *SUT-Gen*

In this section we assess the state-of-the-art IPOG [73] and BOT-its [8, 100] MCAC algorithms that are described in Sections 2.3.1 and 4.1 by using the available MCAC tools and benchmarks generated with the *SUT-G* generator (see Section 5.1)

This evaluation has two main goals:

1. Analyze the impact of the generated SUT constraints with respect to the same unconstrained SUT.

2. 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.

Unfortunately, we did not find any state-of-the-art tool for generating MCACs that supported the *Extended ACTS* format described in Section 5.1.1. Although there are some MCAC tools that could be adapted to support this format (one of them is the ACTS tool [34], in particular, their IPOG implementation that uses a Constraint Programming (CP) solver to handle SUT constraints [104]), these modifications were out of the scope of this work. Instead of that, we decided to adapt our own version of the *IPOG* algorithm (named *CTLOG*) with full support for both the *ACTS* and *Extended ACTS* SUT formats. In this case, constraints are handled using the SAT solver Glucose 4.1 [16].

To ensure that our implementation is competitive with ACTS, we tested both algorithms using the 58 state-of-the-art benchmarks for strengths $t = \{2,3,4\}$, as both implementations support this input format. Table 5.1 shows these results. We executed the ACTS implementation one time and our implementation 10 times with different seeds to mitigate the stochastic behaviour of the algorithm [12].

As we can observe, the *CTLOG* implementation of IPOG is able to solve 9 more instances than *ACTS* for strengths $t = \{2,3,4\}$. In general, our IPOG implementation finds MCACs faster than *ACTS*. Regarding MCAC sizes, both approaches obtain similar results. Therefore, it makes sense to use our own IPOG implementation as we have demonstrated that it is equivalent to or even better than the *ACTS* implementation.

We selected the *BOT-its* and *PBOT-its* [8] algorithms to compare against IPOG. All our algorithms are implemented in Python except for the most critical parts, which have been implemented in Nim [13]. We use the OptiLog [2] Python framework to efficiently use SAT solvers and encoders. In both cases, we use the Glucose 4.1 SAT solver [16]. These algorithms have been adapted to support the new SUT format discussed in Section 5.1.1.

Regarding the generated benchmarks, we decided to use industrial SAT instances for our study. The main idea is that the generated SUT constraints will keep some of the industrial-like structure of the original SAT formula, and can be a good approximation of the constraints of other real-world industrial SUTs.

We selected the instances of the SAT competition 2009 [95] to try to find solvable satisfiable instances for modern SAT solvers more easily [14]. Notice that this selection is only for convenience, as *SUT-G* is able to adapt the hardness of the input instance as explained in Section 5.1.

We also used Glucose 4.1 [16] as incremental SAT solver for Algorithm SUT-G. We set the parameters `rnd-freq` and `rnd-pol-freq` of the solver to 0.5 to allow some variations on the results of the `solve_subproblem` function. We set the constants $\Delta_{\mathcal{A}}$ and $\nabla_{\mathcal{A}}$ to 10 and 5 respectively, $\mathcal{S}$ to five random seeds and MAX_TRIES to 100. Additionally, we limited each query to the SAT solver to 300s, so if one of these queries cannot be completed we consider the formula as UNSAT.

We generated benchmarks with 25, 50 and 100 parameters of domain 2 using a limit c_{max} of 5000 and 10000 conflicts, setting c_{min} always to $c_{max}/2$ (see Section 5.1 for more details regarding the generation parameters). In total, we generated 105 crafted benchmarks that we will make available to the community, as well as the generator.

Our experimentation environment consists of a cluster of nodes with two AMD 7403 processors each (24 cores at 2.8GHz) and 21 GB of RAM per core. Tables 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 show the results of our experimentation. We show in bold the best results in terms of test suite size and time. We run each benchmark with 5 different seeds to mitigate the stochastic behaviour of the tested algorithms, using a time limit of 8h and a memory limit of 21 GB. Additionally, we executed each benchmark for strengths $t = \{2,3,4\}$.

In the following sections we will analyze the results of our experimental evaluation.

[12] The ACTS tool does not expose the *seed* parameter

[13] The Nim programming language (`https://nim-lang.org/`)

[14] Theoretically, SAT solvers that apply modern techniques should be able to solve more efficiently instances of previous SAT competitions.

	$t=2$				$t=3$				$t=4$			
	ACTS		CTLOG		ACTS		CTLOG		ACTS		CTLOG	
	size	time	size	time	size	time	size	time	size	time	size	time
[38]												
1	48	9s	44.9	4s	293	12s	288.3	8s	1680	1207s	1678.6	143s
2	32	9s	34.5	4s	174	6s	181.3	7s	873	586s	883.6	89s
3	19	9s	20.0	4s	71	2s	66.3	4s	212	2s	214.3	4s
4	22	7s	22.6	4s	102	3s	101.4	4s	374	12s	381.2	18s
5	54	10s	51.9	5s	386	177s	378.0	16s	-	-	2437.7	1006s
6	25	8s	26.7	4s	119	5s	120.9	6s	491	116s	492.9	90s
7	12	9s	12.2	1s	35	2s	35.4	1s	93	2s	96.7	3s
8	47	8s	45.7	2s	326	63s	317.3	9s	1988	6819s	1896.8	340s
9	22	9s	21.3	4s	84	4s	83.5	5s	268	77s	273.8	50s
10	47	8s	46.7	3s	329	117s	328.4	14s	2063	17447s	2032.5	1185s
11	47	8s	46.7	4s	318	26s	316.3	8s	1885	2150s	1896.2	149s
12	43	11s	41.1	4s	263	46s	262.0	12s	1465	5676s	1443.8	588s
13	40	10s	38.5	2s	200	10s	207.9	8s	1040	1668s	1042.3	488s
14	39	12s	39.5	4s	244	4s	242.1	7s	1163	225s	1161.7	86s
15	32	9s	32.9	4s	173	4s	174.2	5s	770	21s	774.5	17s
16	25	7s	25.5	4s	117	7s	119.2	8s	453	321s	452.0	146s
17	41	9s	41.0	2s	265	48s	262.2	7s	1514	4904s	1463.5	285s
18	52	9s	47.6	4s	344	27s	321.3	14s	2145	3999s	1990.4	451s
19	51	11s	50.1	5s	373	159s	374.4	21s	-	-	2494.8	2095s
20	60	10s	58.7	3s	463	482s	465.8	23s	-	-	3261.1	2414s
21	39	9s	39.5	2s	235	9s	234.1	6s	1070	494s	1091.6	185s
22	37	9s	36.7	4s	164	5s	165.3	6s	664	100s	669.3	73s
23	14	9s	14.1	4s	48	2s	47.9	4s	140	3s	140.3	5s
24	48	8s	48.7	4s	341	42s	336.8	14s	2105	4512s	2095.6	517s
25	52	7s	53.5	4s	404	58s	403.1	12s	2673	9514s	2652.0	369s
26	34	9s	35.0	4s	207	17s	208.2	8s	1111	1373s	1136.0	151s
27	37	7s	37.8	4s	204	4s	212.0	5s	1004	22s	1017.6	19s
28	57	8s	54.7	5s	420	119s	419.6	23s	-	-	2865.7	2332s
29	29	7s	28.6	4s	154	6s	153.2	8s	681	1354s	713.3	203s
30	22	7s	20.7	4s	100	5s	94.8	6s	386	142s	376.0	76s
apache	33	9s	33.0	5s	173	7s	173.1	10s	838	3468s	838.7	362s
bugzilla	19	7s	18.1	4s	68	2s	67.0	5s	242	4s	229.0	3s
gcc	23	7s	24.6	5s	108	7s	167.3	64s	444	3386s	573.4	6669s
spins	26	6s	25.5	4s	98	2s	114.8	4s	393	3s	426.3	5s
spinv	45	7s	45.4	4s	286	4s	305.9	15s	1631	20s	1818.0	870s
[96]												
Banking1	15	11s	16.2	4s	58	3s	55.4	4s	139	3s	141.8	4s
Banking2	11	11s	11.5	4s	39	2s	39.0	3s	96	2s	95.6	4s
CommProtocol	19	13s	19.7	4s	49	7s	50.4	4s	97	36s	98.3	4s
Concurrency	6	11s	5.4	4s	8	2s	8.0	4s	8	2s	8.0	4s
Healthcare1	30	10s	30.2	4s	105	2s	102.5	4s	341	2s	326.6	4s
Healthcare2	16	11s	16.4	4s	67	2s	64.1	3s	220	4s	222.0	4s
Healthcare3	38	11s	38.4	4s	209	3s	202.9	4s	1004	11s	973.4	9s
Healthcare4	49	12s	48.6	4s	294	3s	292.4	5s	1644	15s	1630.6	13s
Insurance	527	11s	527.9	4s	6866	2s	6930.4	13s	75764	5s	75669.1	99s
NetworkMgmt	112	11s	116.3	4s	1125	2s	1129.9	4s	6267	3s	6248.2	10s
ProcessorComm1	29	11s	28.9	4s	163	2s	159.9	4s	670	3s	668.1	6s
ProcessorComm2	32	13s	31.4	4s	161	65s	160.9	4s	744	7250s	741.2	14s
Services	106	17s	106.4	4s	963	186s	915.1	5s	6855	4718s	6853.7	14s
Storage1	17	11s	17.3	4s	25	2s	25.0	4s	25	2s	25.0	4s
Storage2	18	11s	18.2	4s	74	1s	61.6	4s	195	1s	181.9	4s
Storage3	50	12s	54.5	4s	239	5s	239.2	4s	752	40s	753.4	5s
Storage4	136	10s	133.5	4s	990	2s	1005.7	6s	6636	3s	6589.3	21s
Storage5	218	13s	228.6	4s	1879	6s	1941.7	10s	13292	17s	13454.5	113s
SystemMgmt	17	10s	16.7	4s	60	2s	56.1	3s	152	2s	149.8	4s
Telecom	32	11s	30.7	4s	126	2s	125.4	3s	392	2s	395.2	4s
[102]												
RL-A-mod	155	436s	158.5	4s	-	-	1155.9	13s	-	-	7767.3	336s
RL-B-mod	-	-	761.3	8s	-	-	14201.1	205s	-	-	-	-
[100]												
Company2	81	31s	82.3	1s	424	630s	432.4	2s	-	-	1392.1	5s

TABLE 5.1: Comparison of IPOG implementations in *ACTS* and *CTLOG* tools for strengths $t = \{2, 3, 4\}$

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		bot	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**12.00**	2.07	12.40	**1.06**	30.10	**1.30**	**29.70**	1.40	**73.90**	**1.50**	76.10	9.69
AProVE09-01	**10.00**	11.33	10.20	**9.54**	22.40	15.53	**21.80**	**11.19**	**54.20**	26.06	55.20	**18.12**
AProVE09-05	**8.40**	12.18	10.00	**10.58**	**16.60**	14.74	18.60	**13.39**	**23.60**	21.13	24.60	**18.49**
AProVE09-07	**3.00**	9.99	**3.00**	**8.46**	**3.00**	10.64	**3.00**	**9.30**	**3.00**	16.21	**3.00**	**12.90**
AProVE09-08	**2.00**	**9.09**	**2.00**	9.13	**2.00**	**9.76**	**2.00**	9.89	**2.00**	12.61	**2.00**	**12.13**
AProVE09-17	**8.20**	24.65	10.40	**23.41**	**18.60**	32.45	24.20	**32.20**	**43.00**	52.07	52.20	**48.40**
AProVE09-20	**16.00**	**143.69**	24.00	245.39	**47.80**	**191.92**	77.40	327.50	**130.60**	**352.73**	185.80	385.19
AProVE09-21	**21.40**	304.04	33.00	**161.28**	**77.00**	741.11	137.20	**552.50**	**245.40**	2111.51	441.00	**1348.85**
AProVE09-24	**13.00**	27.53	19.60	**21.01**	**42.00**	47.59	55.80	**29.13**	**120.80**	150.69	136.20	**51.08**
gss-14	**1.00**	**98.04**	**1.00**	110.95	**1.00**	**140.92**	**1.00**	247.86	**1.00**	**164.23**	**1.00**	400.54
q_query_3_L60	**2.00**	52.19	**2.00**	**47.35**	**2.00**	58.76	**2.00**	**45.50**	**2.00**	93.62	**2.00**	**53.59**
q_query_3_l37	**13.00**	20.04	15.80	**14.85**	**39.00**	28.83	41.40	**17.37**	**99.00**	54.76	102.40	**28.17**
q_query_3_l38	**10.60**	14.15	11.00	**10.40**	28.80	18.79	**28.40**	**11.92**	**68.60**	29.98	70.80	**17.26**
q_query_3_l39	**10.20**	12.91	10.60	**11.06**	**26.00**	17.19	26.20	**12.20**	**62.80**	32.05	65.00	**17.39**
q_query_3_l40	**10.00**	18.09	10.60	**15.46**	**24.40**	21.06	25.00	**17.34**	**59.80**	30.35	61.20	**24.93**
q_query_3_l41	**10.40**	**20.13**	13.60	21.46	**29.20**	25.63	32.60	**24.66**	**77.60**	43.71	79.80	**31.10**
q_query_3_l42	**9.20**	22.58	10.40	**22.43**	**22.60**	25.64	23.20	**25.40**	**55.80**	34.54	56.40	**26.82**
q_query_3_l43	9.00	18.04	**8.80**	**14.79**	21.80	20.40	**21.00**	**14.67**	50.60	27.81	**50.20**	**19.68**
rbcl_xits_14	**8.20**	72.41	11.40	**49.86**	**19.40**	123.42	23.60	**44.50**	**37.80**	108.83	42.00	**96.74**
rbcl_xits_14-1	**8.20**	72.45	11.40	**49.99**	**19.40**	120.37	23.60	**46.51**	**37.80**	104.15	42.00	**96.50**
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-
rpoc_xits_17-1	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.2: Test suite size and runtime for the instances with 25 parameters and 5000 conflicts

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		bot	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**12.00**	2.07	12.40	**1.06**	30.10	**1.30**	**29.70**	1.40	**73.90**	**1.50**	76.10	9.69
AProVE09-07	**3.00**	9.27	**3.00**	**7.95**	**3.00**	13.04	**3.00**	**10.94**	**3.00**	19.00	**3.00**	**13.87**
AProVE09-08	**5.00**	23.99	**5.00**	**19.05**	**6.00**	**25.87**	**6.00**	27.21	**6.00**	35.57	**6.00**	**23.03**
AProVE09-15	**11.00**	82.52	20.80	**48.83**	**27.20**	144.29	48.60	**78.44**	**76.00**	319.77	92.60	**103.55**
AProVE09-20	**21.60**	**247.92**	33.60	261.64	**71.40**	**359.70**	111.40	429.13	**197.40**	661.52	292.40	**638.34**
UR-10	**6.00**	38.68	6.20	**35.13**	**8.20**	**40.39**	10.20	41.61	**16.00**	63.62	**16.00**	**42.41**
gss-14	**1.00**	143.38	**1.00**	**101.77**	**1.00**	175.34	**1.00**	**154.93**	**1.00**	**260.89**	**1.00**	366.99
q_query_3_l37	**11.40**	40.40	13.40	**18.87**	**35.00**	69.35	37.00	**27.01**	**92.80**	115.16	100.00	**42.81**
q_query_3_l38	**10.00**	21.16	11.60	**13.94**	**26.40**	28.51	29.00	**16.79**	**70.00**	43.00	73.80	**24.17**
q_query_3_l39	**10.80**	18.30	12.20	**16.08**	**27.00**	21.29	30.40	**16.83**	**70.00**	35.90	71.40	**23.42**
q_query_3_l40	**12.80**	45.53	15.80	**38.42**	**36.20**	57.04	42.80	**48.12**	107.40	80.07	**107.00**	**56.26**
q_query_3_l41	**9.80**	23.39	11.20	**22.60**	**24.80**	29.62	26.40	**27.30**	**62.40**	46.15	64.20	**28.15**
q_query_3_l42	8.00	**25.12**	**7.80**	29.08	**16.40**	30.46	16.80	**26.68**	**35.20**	**34.51**	37.40	35.00
q_query_3_l43	10.00	13.80	**9.80**	**11.68**	**24.00**	16.62	24.20	**11.44**	**56.80**	23.57	58.60	**17.33**
rbcl_xits_14	**8.80**	3515.88	12.20	**1654.27**	**22.80**	8542.25	33.75	**2206.95**	**57.25**	8448.56	76.80	**4854.20**
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.3: Test suite size and runtime for the instances with 25 parameters and 10000 conflicts

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		bot	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**14.00**	1.24	15.40	**1.07**	39.20	**1.23**	**39.00**	4.67	**101.10**	**1.99**	101.20	230.84
AProVE09-01	**12.00**	14.69	13.00	**9.01**	**31.20**	27.47	32.00	**14.10**	**79.80**	**81.84**	82.00	96.79
AProVE09-05	**18.20**	16.41	20.40	**15.57**	**39.00**	30.69	42.00	**20.66**	**51.60**	247.81	54.40	**148.37**
AProVE09-07	**5.00**	**9.81**	**5.00**	9.82	**5.00**	15.79	**5.00**	**12.77**	**5.00**	130.74	**5.00**	**87.56**
AProVE09-08	**2.00**	**8.15**	**2.00**	8.23	**2.00**	11.12	**2.00**	**10.50**	**2.00**	**47.45**	**2.00**	67.34
AProVE09-17	**15.20**	53.27	26.60	**40.34**	**55.60**	142.20	100.60	**110.45**	**199.00**	730.10	306.20	**542.11**
AProVE09-20	**37.60**	**206.62**	61.20	251.94	**168.80**	607.08	255.40	**397.35**	**670.00**	8655.74	863.40	**1579.16**
AProVE09-21	**48.40**	568.46	92.20	**348.80**	**266.00**	3485.15	584.60	**3215.69**	**1225.00**	20551.41	2454.40	**11135.27**
AProVE09-24	**21.60**	43.64	31.20	**27.10**	**88.40**	188.11	103.80	**58.96**	**338.80**	2537.30	362.20	**527.29**
gss-14	**1.00**	**103.44**	**1.00**	126.74	**1.00**	**119.52**	**1.00**	265.68	**1.00**	**162.76**	**1.00**	463.80
q_query_3_L60	**7.20**	**57.50**	10.80	96.95	**14.20**	**74.70**	20.40	128.28	**27.80**	**173.44**	31.40	251.91
q_query_3_l37	**18.00**	26.64	23.80	**17.56**	**66.40**	68.08	78.60	**29.48**	**218.00**	415.56	228.00	**178.33**
q_query_3_l38	**17.20**	19.17	20.40	**11.74**	**62.40**	50.78	66.80	**17.69**	**216.80**	246.18	223.60	**139.10**
q_query_3_l39	**12.40**	16.94	14.20	**12.57**	**36.00**	34.06	41.00	**15.20**	**104.80**	211.00	118.80	**137.61**
q_query_3_l40	**11.40**	19.33	12.80	**17.40**	**34.20**	32.73	35.80	**18.63**	**94.80**	156.48	104.00	**110.74**
q_query_3_l41	**14.00**	26.66	17.80	**20.73**	**48.80**	54.39	54.40	**30.10**	**163.60**	267.29	172.60	**175.98**
q_query_3_l42	**11.40**	29.71	**11.40**	**25.04**	**29.80**	41.17	30.80	**23.00**	**83.40**	**111.42**	91.00	111.47
q_query_3_l43	11.60	18.97	**11.40**	**14.55**	35.80	33.80	**34.60**	**18.22**	**90.40**	**91.49**	95.80	93.42
rbcl_xits_14	**11.00**	100.32	21.60	**60.70**	**34.20**	140.27	55.20	**49.82**	**82.40**	**207.96**	123.60	237.65
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.4: Test suite size and runtime for the instances with 50 parameters and 5000 conflicts

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		bot	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**14.00**	1.24	15.40	**1.07**	39.20	**1.23**	**39.00**	4.67	**101.10**	**1.99**	101.20	230.84
AProVE09-07	**6.00**	13.92	6.20	**12.57**	**8.00**	17.85	**8.00**	**11.11**	**8.00**	**95.15**	**8.00**	99.85
AProVE09-08	**6.40**	**22.28**	6.60	27.18	**8.00**	**27.55**	**8.00**	28.67	**8.00**	117.15	**8.00**	**100.95**
AProVE09-15	**19.40**	162.12	36.00	**102.05**	**75.40**	511.28	156.40	**285.90**	**270.00**	3099.48	437.20	**1091.52**
AProVE09-20	**35.20**	**334.61**	58.40	377.62	**168.40**	791.53	268.00	**663.72**	**737.80**	12134.04	1003.80	**2334.18**
UR-10	**7.20**	43.48	8.40	**40.72**	**13.80**	51.24	16.20	**48.60**	**26.20**	144.98	28.40	**116.96**
gss-14	**1.00**	**124.86**	**1.00**	221.15	**1.00**	**185.78**	**1.00**	197.57	**1.00**	**237.18**	**1.00**	308.75
gss-15	**1.00**	**42.59**	**1.00**	77.40	**1.00**	**59.96**	**1.00**	76.36	**1.00**	**99.12**	**1.00**	207.27
q_query_3_l37	**23.80**	57.71	32.20	**29.65**	**111.60**	179.69	129.20	**62.27**	**406.20**	1613.19	461.60	**328.03**
q_query_3_l38	**16.60**	28.07	19.80	**17.39**	**63.40**	62.65	75.40	**24.72**	**226.60**	269.64	241.80	**170.71**
q_query_3_l39	**15.60**	23.31	18.00	**16.68**	**53.80**	47.27	57.80	**20.76**	**186.00**	322.07	189.20	**141.60**
q_query_3_l40	**20.80**	**55.32**	25.80	57.08	**85.60**	115.03	92.20	**67.61**	**272.60**	569.56	284.40	**269.98**
q_query_3_l41	**13.20**	28.55	14.60	**27.47**	**41.60**	50.95	43.80	**26.81**	**123.00**	204.09	136.40	**141.97**
q_query_3_l42	**10.60**	**29.34**	11.80	34.17	**27.80**	41.61	28.20	**37.92**	**72.60**	**88.61**	79.20	107.75
q_query_3_l43	**11.80**	18.23	12.80	**11.54**	35.40	31.77	**33.60**	**12.79**	**92.20**	95.67	98.20	**87.13**
rbcl_xits_14	**11.33**	9438.27	22.20	**3470.61**	**43.25**	8778.63	68.50	**8434.12**	**105.00**	14957.99	181.20	**7680.59**
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.5: Test suite size and runtime for the instances with 50 parameters and 10000 conflicts

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		pbot1G	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**16.00**	**1.31**	17.10	1.36	48.30	**1.75**	**48.10**	39.29	**131.40**	**16.40**	240.20	1575.46
AProVE09-01	**13.60**	19.86	14.20	**5.94**	**40.80**	61.75	43.80	**34.94**	**118.20**	**693.01**	145.60	1165.26
AProVE09-05	**26.80**	15.74	29.60	**15.63**	**53.00**	117.18	54.80	**70.81**	**71.20**	3405.83	75.40	**1272.09**
AProVE09-07	**5.00**	**5.98**	**5.00**	6.64	**5.00**	53.24	**5.00**	**31.71**	**5.00**	3119.65	**5.00**	**1078.51**
AProVE09-08	**4.00**	**4.31**	**4.00**	4.87	**4.00**	**15.52**	**4.00**	24.79	**4.00**	**628.45**	**4.00**	1092.38
AProVE09-17	**35.00**	86.08	59.40	**66.98**	**192.00**	1535.24	270.00	**357.02**	-	-	**1027.80**	**2482.75**
AProVE09-20	**66.80**	307.55	108.20	**276.30**	**462.20**	4036.59	597.40	**964.15**	-	-	**3065.20**	**9220.35**
AProVE09-21	**100.00**	898.98	191.60	**654.85**	**893.60**	24042.60	1784.80	**9812.63**	-	-	-	-
AProVE09-24	**42.40**	120.59	51.20	**54.57**	**224.40**	2401.02	239.00	**355.86**	-	-	**1154.00**	**3983.49**
gss-14	**1.00**	**108.34**	**1.00**	133.56	**1.00**	**155.89**	**1.00**	312.83	**1.00**	**785.80**	**1.00**	1537.03
q_query_3_L60	**10.00**	70.75	13.20	**58.36**	**27.80**	103.27	37.80	**99.90**	**72.60**	**1200.75**	85.40	1278.38
q_query_3_l37	**34.00**	41.02	45.80	**21.80**	**173.00**	416.29	187.60	**100.40**	**680.75**	13673.49	728.60	**1987.41**
q_query_3_l38	**27.20**	29.13	33.20	**11.03**	**129.20**	288.13	132.40	**61.04**	**531.00**	12746.11	550.00	**1371.29**
q_query_3_l39	**17.80**	25.30	18.40	**10.98**	**67.40**	183.52	70.80	**54.98**	**240.20**	5492.51	272.60	**1187.12**
q_query_3_l40	**16.60**	23.18	19.20	**13.67**	**64.00**	132.17	67.20	**43.66**	**236.00**	5006.47	241.40	**1093.34**
q_query_3_l41	**22.60**	36.09	26.60	**22.28**	**98.00**	224.07	99.80	**62.51**	433.00	4265.88	**411.00**	**1298.78**
q_query_3_l42	**14.20**	30.25	14.60	**21.96**	**50.60**	78.71	54.00	**50.37**	**185.60**	1113.71	192.40	**1052.16**
q_query_3_l43	**17.40**	25.14	**17.40**	**12.09**	81.80	127.90	**66.20**	**47.24**	231.20	2203.31	**230.60**	**1122.33**
rbcl_xits_14	**14.80**	92.34	34.20	**67.15**	**63.00**	139.40	93.00	**81.29**	**173.00**	**842.88**	258.40	1679.15
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.6: Test suite size and runtime for the instances with 100 parameters and 5000 conflicts

	$t = 2$				$t = 3$				$t = 4$			
	ipog		bot		ipog		bot		ipog		pbot1G	
	size	time	size	time	size	time	size	time	size	time	size	time
unconstr	**16.00**	**1.31**	17.10	1.36	48.30	**1.75**	**48.10**	39.29	**131.40**	**16.40**	240.20	1575.46
AProVE09-07	**11.40**	10.65	14.00	**10.11**	**24.20**	**36.28**	25.60	59.94	**31.00**	**919.37**	33.40	1177.58
AProVE09-08	**7.00**	**19.58**	8.00	28.75	**10.20**	**42.11**	10.40	57.62	**12.00**	1185.18	**12.00**	**1087.00**
AProVE09-15	**50.00**	378.53	76.20	**210.04**	**326.60**	12624.17	440.40	**1012.74**	-	-	**2027.00**	**6279.95**
AProVE09-20	**61.60**	**387.18**	97.80	478.77	**457.80**	8122.76	627.60	**1236.00**	-	-	**3380.20**	**10364.28**
UR-10	**8.20**	**46.61**	10.00	47.52	**16.40**	93.37	20.60	**69.26**	**38.80**	1957.95	48.80	**1310.88**
gss-14	**1.00**	**194.64**	**1.00**	537.10	**1.00**	**241.69**	**1.00**	466.77	**1.00**	**854.17**	**1.00**	1621.46
q_query_3_l37	**48.20**	159.63	55.20	**105.19**	**272.40**	2027.72	339.60	**350.08**	-	-	**1447.80**	**3708.98**
q_query_3_l38	**31.20**	46.08	37.20	**16.22**	**159.20**	406.82	172.80	**81.76**	**709.25**	11638.60	715.00	**1758.13**
q_query_3_l39	**22.00**	31.87	25.40	**15.82**	90.20	213.85	**87.80**	**50.49**	371.80	3670.82	**348.00**	**1189.03**
q_query_3_l40	**34.00**	79.07	46.40	**75.93**	190.20	735.57	**183.80**	**143.70**	**635.33**	12984.01	641.00	**1956.74**
q_query_3_l41	**17.80**	38.95	24.40	**33.39**	**71.00**	161.96	84.60	**76.65**	**248.40**	3405.22	304.80	**1294.24**
q_query_3_l42	**12.40**	**28.19**	13.60	29.93	**42.20**	64.01	44.20	**48.39**	**133.80**	**733.06**	155.00	1079.87
q_query_3_l43	**14.80**	20.69	15.80	**9.28**	58.60	87.31	**51.80**	**35.39**	**170.40**	1900.25	181.40	**1018.44**
rbcl_xits_14	**21.60**	4482.73	62.80	**3014.47**	**88.60**	7630.25	222.50	**3060.48**	**293.50**	18368.60	627.40	**5589.34**
rpoc_xits_17	-	-	-	-	-	-	-	-	-	-	-	-

TABLE 5.7: Test suite size and runtime for the instances with 100 parameters and 10000 conflicts

5.2.1 Impact of the SUT constraints

The first question we address is how these SUT constraints impact the performance of the tested algorithms. To analyze this, we included an execution of a benchmark with the same number of parameters but without SUT constraints on the *unconstr* row of Tables 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7.

In general, we observe that the addition of SUT constraints has a negative impact on the generation runtime of the MCAC, regardless of the number of parameters of the SUT, the strength t or the MCAC algorithm. This is expected, as at some point we need to ensure that the test case that we are building is consistent with the SUT constraints. This negative effect is amplified when we increase the number of parameters, strength or *hardness* of the SUT constraints (i.e. number of conflicts).

However, we found some cases where the runtime is reduced, especially in some executions for $t = 4$ on the BOT-its and PBOT-its algorithms (for example, benchmark *AProVE09-08* in Table 5.4 for $t = 4$ and BOT-its, or benchmark *q_query_3_l43* in Table 5.7 for $t = 4$ and PBOT-its). This suggests that some of the efficiencies on the BOT-its algorithms regarding SUT constraints handling might benefit its performance in some circumstances.

Regarding the test suite size, we observe more variation than with runtime. In some cases it increases (for example benchmark *AProVE09-21* in Table 5.4 for all the strengths and algorithms), while in others it decreases (for example benchmark *rpoc_xits_14* in Table 5.3 for $t = 3$ in the IPOG algorithm). In the first case, there might be some combinations that belong to some tests of the unconstrained version that are now forbidden, so these tests need to be *splitted* into different tests, increasing the overall test suite size. For the second case, the generated benchmark is so constrained that it is forbidding some of the possible tests that we can find in the unconstrained version, reducing the final test suite size. An extreme example of that is benchmark *AProVE09-07* in Table 5.6. Its test suite size is always 5 for all the strengths and algorithms, which suggests that there are only 5 possible models for the SUT constraints.

Notice that the *IPOG* and *BOT-its* algorithms cannot guarantee the optimality of the reported test suite (i.e. they cannot certify the *CAN*), so solutions with smaller test suites may exist. Nonetheless, these results provide interesting insights about the behaviour of practical MCAC generation algorithms when dealing with *hard* SUT constraints, as complete approaches that can obtain optimal MCACs such as MaxSAT MCAC [15, 5] or the CALOT [101] algorithm will struggle with these kinds of benchmarks (see Section 5.3 for more in-depth discussion about this).

5.2.2 Comparing the performance of the IPOG and (P)BOT-its algorithms

In Section 5.2.1 we analyzed the overall impact of the generated benchmarks over their unconstrained version. In this section we analyze the differences on performance for the *IPOG* and *(P)BOT-its* algorithms (see Sections 2.3.1 and 4.1 respectively).

In general, we observe how *IPOG* obtains better sizes than *BOT-its* but worse run times. As we discussed in previous sections, the *BOT-its* algorithm heuristically creates one test at a time that might be inconsistent during its construction, and later *amends* it to try to reduce the number of SAT solver queries. It is clear that this procedure reduces the efficiency of the heuristic and therefore can increase the final test suite size, as some of its choices might be suboptimal once the test case is repaired. On the other hand its final run time is reduced, especially when dealing

with *hard* SUT constraints, as SAT solver queries are also reduced with respect to *IPOG*.

This is an interesting result, as in previous works the *IPOG* algorithm obtained better run times than *BOT-its* on the original state-of-the-art benchmarks [100, 8].

In Tables 5.6 and 5.7, where we are solving SUT models with 100 parameters, we observe the limitations in terms of memory consumption of both *IPOG* and *BOT-its*. When we increase the strength t above 3, *IPOG* is only able to report an MCAC for 25 of the 34 benchmarks, while the basic *BOT-its* could not report any of them[15]. This is why we executed the *pool* version of the *BOT-its* algorithm with a pool of 1GB (referred as *pbot1G* in the mentioned tables, see [8]). In contrast, this algorithm can solve all benchmarks except 3.

In these cases, we observe larger differences in runtime between IPOG and PBOT-its. In particular we observe a difference of an order of magnitude for benchmarks *q_query_3_l37* and *q_query_3_l38* in Table 5.6, and *q_query_3_l38* and *q_query_3_l40* in Table 5.7 for $t = 4$.

In contrast, we observe a slightly increase in the test suite size in PBOT-its for these benchmarks, compared to IPOG. We noticed other benchmarks where this gap in terms of test suite size is further amplified (*AProVE09-21* for $t = 3$ in Table 5.2, *AProVE09-15* for $t = \{2, 3\}$ in Table 5.3 and *AProVE09-21* for $t = 2, 3, 4$ in Table 5.4 are some examples where (P)BOT-its almost doubles the test suite size obtained by IPOG).

Finally, we observe that there is a particular benchmark that is never solved (benchmark *rbcl_xits_17* in Tables 5.2, 5.3, 5.4, 5.5 and 5.7). Although we tried to adapt the hardness of the generated benchmarks in Algorithm SUT-G, there might be cases where this is not always possible. In particular, it seems that in this benchmark the algorithms try to explore an exponential branch of the formula when completing certain test cases. Notice that the *IPOG* algorithm would not be able to avoid these branches. On the other hand, *BOT-its* could be modified to try to mitigate this effect by adapting the *amend* procedure on each test. This can be considered future work.

Thanks to the development of the *SUT-G* generator, we have been able to provide another point of view on the performance of two well-known MCAC generation algorithms. As result, in the next section, we will try to provide some recipes to effectively apply MCAC algorithms in real-world scenarios.

5.3 Recipes for Choosing MCAC Tools

In this section, we provide several interesting insights about the application of MCAC generation tools. We extracted them thanks to the experimental evaluation that we conducted using the *SUT-G* generator instances (see Section 5.1). These instances provided another point of view on the behaviour of the tested algorithm with respect to the currently available CT benchmarks. Notice that in this case, we focus on the MCAC problem, but other CT problems can also be considered for this analysis.

In [8, 100], authors show how IPOG is faster than the BOT-its/*Algorithm 5* algorithm for the current CT benchmarks. However, as we discussed in Section 5.2.2, we observe now, that in general, the BOT-its algorithm is able to obtain an MCAC in less time than IPOG. Notice that IPOG needs to ensure at each point that all the partially-built test cases are consistent with the SUT constraints (see Section 2.3.1).

[15] We did not include these results in the tables. Instead, we just report the results of PBOT-its

This implies a query to the SAT solver for each value that it fixes in horizontal extension, another query to the SAT solver for each tuple that has not been covered and an additional query for each of the tests where a tuple can be covered until it is covered.

In contrast, the BOT-its algorithm only checks the first tuple that it fixes in the test and once it finishes building it[16] (see Section 4.1). In case it is not, it removes the last fixed parameter and checks the test again, until the test is consistent. Although this method can produce as many SAT queries as the IPOG algorithm in the worst case, we observe that this is not true for the average case. Therefore, the BOT-its algorithm can potentially reduce the number of SAT queries with respect to IPOG. This comes with the drawback of having a more incomplete heuristic than IPOG, which can increase the test suite size. Additionally, the BOT-its algorithm is able to eliminate forbidden tuples more efficiently than IPOG (i.e. it does not require one SAT query for each forbidden tuple).

On the other hand, IPOG obtains better MCAC sizes in general. This is also observed in [8] for the current CT benchmarks.

As rule of thumb, we recommend using the IPOG algorithm for SUTs with *simple* constraints and low strength t (which are all the available CT benchmarks until this work). To try to obtain smaller test suites for these instances, metaheuristic approaches such as [48, 81] can also be useful. These approaches take an already-built MCAC (for example, one obtained using IPOG) and try to eliminate test cases by swapping values in the test suite using local search techniques. For the cases where an optimal MCAC is required there exist several algorithms such as MaxSAT MCAC [15, 5] or CALOT [101]. Notice however that obtaining a minimal MCAC for a given SUT and strength t is an NP-Hard [82] problem, and that only relatively small instances with small strengths (e.g. $t = 2$) are suitable for these methods.

In case the generation time is important, for more complex constraints we recommend the BOT-its algorithm and its variants. Additionally, for higher strengths where the memory consumption of the IPOG algorithm is too high, the PBOT-its variant of BOT-its can mitigate these memory issues. As a side note, notice also that BOT-its and PBOT-its allow *online* testing of the SUT, as test cases can be applied to the system as soon as they are produced. In the IPOG algorithm, a test case is not completed until the algorithm finishes.

Trying to find optimal MCACs for this kind of instances by using MaxSAT MCAC or CALOT can be more challenging. Unlike in IPOG or BOT-its, these other algorithms need to encode a copy of the SUT constraints for each test, making these approaches impractical when having large SUT constraints (even for $t = 2$).

Similarly, metaheuristic approaches can also suffer when dealing with *hard* SUT constraints, as for each change that they perform in the test suite, they must ensure that it is consistent with the SUT constraints by querying the SAT solver.

[16] For each of the fixed parameters, BOT-its performs a *limited* query to the SAT solver, which is much more efficient in terms of time

Chapter 6

OptiLog v2: Model, Solve, Tune And Run

6.1 Introduction

In the last twenty years, the efficiency of SAT engines (solvers) has experimented a great success. Actually, they have become the core engines of other higher-level engines: #SAT (Sharp-SAT), MaxSAT (Maximum Satisfiability), QBF (Quantified Boolean Formulas), PBO (Pseudo-Boolean Optimization), SMT (Satisfiability Modulo Theories), Model finding, Theorem proving, ASP (Answer Set Programming), LCG (Lazy Clause Generation), CSP (Constraint Satisfaction Problems), etc.

Despite the tremendous success of SAT applications in several domains, the access to these resources by members of other research communities, industrial environments, and students of undergraduate courses has been rather limited due to the absence of friendly frameworks. The same story applies to other areas of computer science.

The Python programming language [98], thanks to its simplicity, has dramatically turned the situation around, becoming the middleware to interconnect many scientific libraries through Python bindings such as Numpy [62], Pandas [99], scikit-learn [92], Pytorch [91], Keras [37], etc. This interconnection has definitely allowed to develop more complex applications and to indirectly justify further the individual utility of each library.

In Constraint Programming we also find several Python applications or bindings such as CPLEX [66], Gurobi [61], OR-Tools [58], COIN-OR [39], SCIP [52], Z3 [42], PySMT [54], *cnfgen* [71], PySAT [68], PyPbLib [80], SAT Heritage [17], OptiLog [7], etc.

In this paper, we focus on a new release of the OptiLog Python framework [7] for SAT-based applications with significant contributions. The idea is to make of OptiLog the tool of choice to support researchers, practitioners, or students along all the development process that involves modelling the problem, implementing a solving approach, tuning the overall approach, and evaluating its effectiveness. OptiLog covers all these steps. Moreover, thanks to its versatility, it can be used to develop other systems not necessarily related to SAT problems.

The paper is structured as follows: in Section 6.2 we present the general architecture of the new OptiLog framework. In Section 6.3, we present the new *Modelling* module to define problems. In Section 6.4 we describe how the *Solvers* module has been completely redesigned. Next, in section 6.5 we introduce the new *BlackBox* module to execute external tools within OptiLog. In Section 6.6 we describe the new additions to the *Tuning* module. Finally, in Section 6.7 we describe the new *Running* module to execute experiments and parse the results.

We provide in OptiLog's documentation [79] a case study with full detail that covers all the steps of the development process.

6.2 OptiLog Framework Architecture

OptiLog [7] was designed as a Python library for rapid prototyping of SAT-based systems. However, we will see that it provides features (such as the running and tuning modules) that can be used in other scenarios not necessarily only for developing constraint programming systems. In particular, OptiLog provided four main modules for its end-user API: The Formula module, the PB Encoder module (renamed as *Encoder* Module), the SAT solver module, and the Automatic Configuration (AC) module (renamed as *Tuning* module).

In this paper, we extend OptiLog's end-user API to ease its usage in education and industry by providing three new modules: a higher-level modelling language within the *Modelling* module, a *Running* module that simplifies the execution of experiments and their analysis, and a *BlackBox* Module that eases the integration of third-party tools. Additionally, the original *Solvers* module, which allows executing within OptiLog the C++ libraries of incremental SAT solvers, has been completely redesigned. Also, the *Tuning* module that allows the interconnection with automatic configurators has been extended and integrated with the new *Running* module.

OptiLog is the evolution of our PyPbLib [80] package, which is also used by PySAT [68]. OptiLog is now also available through PyPi [51]:

```
$ pip install optilog
```

Figure 6.1 shows the new architecture we propose for OptiLog, a full description on the current architecture can be found in the OptiLog manual [79]. This architecture supports the user along the development process:

1. **Model the problem** into a more richer and compact formalism (combining Non-CNF Boolean and Pseudo-Boolean expressions) through the *Modelling* module. And, translate the model into a formalism supported by constraint programming solvers, e.g. Boolean formulas in Conjunctive Normal Form (CNF) or Weighted CNF, thanks to the *Formula* and *Encoders* modules.

2. **Implement the solving algorithmic approach** through: (i) the *Solvers* module that allows using *External Libraries* such as libraries of incremental C++ SAT solvers and, (ii) the *BlackBox* module to execute *External Tools* such as generators, preprocessors, feature extractors, binaries of other solvers (such as the SAT solvers available from SAT Heritage [18]), or any other system command, etc.

3. **Tune both the encoding and solving choices** from a holistic point of view, through the *Tuning* module, to increase the effectiveness of their interconnection on a given training set.

4. **Evaluate the resulting system** on a test set thanks to the *Running* module.

In the following, we describe with further detail the new contributions.

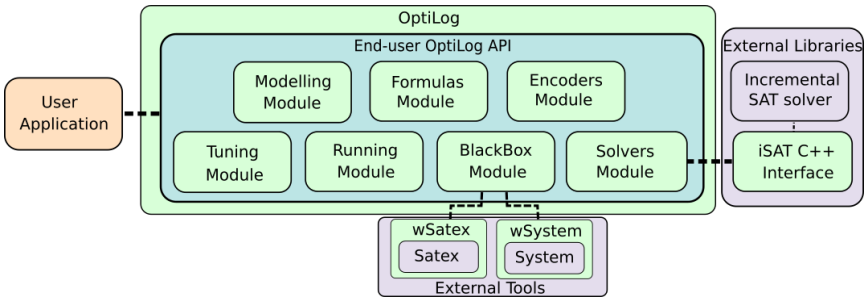


FIGURE 6.1: OptiLog's architecture.

6.3 Modelling Module

This module provides a rich and compact formalism to model problems. In particular, this module allows modelling problems with non-CNF Boolean and Pseudo Boolean expressions that can be automatically transformed into the SAT formula provided by the *Formulas* module. The non-CNF expressions are translated into SAT using the Tseitin transformation. The Pseudo-Boolean expressions are normalized [1] translated into SAT with the additional use of the *Encoders* Module. The goal is to come up with a richer formalism, that frees the user from reimplementing typical transformations to SAT, yet close enough to the formalism accepted by the solvers in OptiLog. We think that OptiLog can be further used by other Tools with higher formalism such as Minizinc [89], i.e, sort of *OptiLog FlatZinc*.

```
1 a = Bool('a')
2 b = Bool('b')
3 c = Bool('c')
4 e1 = ~a + ~b + ~c < 2
5 e2 = ~(a & b & c)
6 e3 = e1 & e2
7 e4 = If(a, b ^ c)
8 p1 = Problem(e1, name='p1')
9 p2 = Problem(e2, name='p2')
10 p3 = Problem(e3, name='p3')
11 p4 = Problem(e4, name='p4')
12 t = TruthTable(p1, p2, p3, p4)
13 t.print()
```

FIGURE 6.2: Basic example of a problem definition.

a	b	c	p1	p2	p3	p4
0	0	0	0	1	0	1
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	0	0	0

FIGURE 6.3: Truth table representation for *p1*, *p2*, *p3* and *p4*

In Figure 6.2, we can see a little example of the new modelling formalism. We first define the Boolean variables that will appear in the formula (lines 1-3). These variables have to be labelled with an identifier.

Then, in line 4 we create our first expression to encode the constraint $\neg a + \neg b + \neg c < 2$. In general, we can directly use the Python logic operators ($\sim$, &, |, $\hat{}$) and their counterparts (`Not`, `And`, `Or`, `Xor`) to create Boolean expressions and the Python

arithmetic operators $(+, -, ^*, <, <=, >=, >, ==)$ to create Pseudo-Boolean expressions. We can also use the `If`, `Iff` classes to create implications and double implications[1]. Lines 5 and 7 encode the constraints $\neg(a \wedge b \wedge c)$ and $a \rightarrow (b \oplus c)$ respectively, whereas in line 6 we encode the conjunction of expressions `e1` and `e2`.

Finally, in lines 8-11 we transform the created expressions to instances of the class *Problem*. A *Problem* represents the conjunction of a set of expressions. In this case, we add a single expression to each *Problem*, and we name each of the problems to reference them later.

In line 12, as an example of how this package can be used for also for educational purposes, we create the truth table for our four problems and we print them in line 13 producing the output shown in Figure 6.3. This is very useful not only to teach any introductory course on propositional logic but also to double-check some small formulas.

Lines 14-19 in Figure 6.4 show how we can use a SAT solver to obtain a solution for our problem. First of all, we need to translate our formula into CNF DIMACS format [43] which is the input format for SAT solvers. In line 14, we create an instance of the SAT solver *Glucose41*. Then, in line 16, we add the clauses forming our CNF formula to the SAT solver and execute the solver in line 17. If the input instance is satisfiable we can obtain a model and decode that model according to the labels of our variables. The resulting model is finally printed in line 19 obtaining the output: `P3 solution: [a, b, ~c].`

```
13 (...)
14 s = Glucose41()
15 p3_cnf = p3.to_cnf_dimacs()
16 s.add_clauses(p3_cnf.clauses)
17 s.solve()
18 solution = cnf.decode_dimacs(s.model())
19 print('P3 solution:', solution)
```

FIGURE 6.4: Example on how to solve *p3* and extract its model.

Now, we can also query whether problem *p*4 is a logic consequence of *p*3 (*p*3 entails *p*4), i.e., $\neg a + \neg b + \neg c < 2, \neg(a \wedge b \wedge c) \models a \rightarrow (b \oplus c)$ which is equivalent to ask whether the conjunction of all the premises and the negation of the consequence, i.e., $(\neg a + \neg b + \neg c < 2) \wedge \neg(a \wedge b \wedge c) \wedge \neg(a \rightarrow (b \oplus c))$ is unsatisfiable. Figure 6.5 shows how to do it in OptiLog.

```
19 (...)
20 s = Glucose41()
21 p5_cnf = Problem(e3 & ~e4).to_cnf_dimacs()
22 s.add_clauses(p5_cnf.clauses)
23 print('Is p5 Satisfiable:', s.solve())
```

FIGURE 6.5: Logic consequence example.

Since the logic consequence is valid the SAT solver reports the formula is unsatisfiable: `Is p5 Satisfiable: False`.

In the OptiLog documentation[79], we can find a more complex application of OptiLog to model, solve, tune and run the SlitherLink problem[2].

[1] These two classes do not naturally map to any Python operator.

[2] Also in the appendix attached to this submission.

There exist other modelling python libraries that can be used to model problems. For example, CPMpy [60] is a library that allows the representation of matrix-related constraints using Numpy arrays [62]. pyAiger is a circuit-oriented modelling library to model combinatorial circuits. Although it shares some similarities with OptiLog's *Modelling* module, it lacks some of its higher-level features such as Pseudo-Boolean expressions support. As future work, we will integrate these libraries into OptiLog to be used within the *Modelling* module.

6.4 Solvers Module

The solvers module has been completely redesigned with a more modular philosophy that allows dynamic solver loading and a more flexible compilation pipeline:

1. **Release of the iSAT interface:** Within this version the python framework OptiLog is able to use *External Libraries* provided they implement a given interface. In particular, OptiLog releases the iSAT C++ interface to the developer community [77] and welcomes solver developers to make their C++ SAT solvers compliant with this interface. Now, a SAT solver compiled with the interface as a shared object (.so file) can be dynamically loaded into OptiLog. At [77] GitHub repository several examples of solvers implementing the iSAT interface can be found.

2. **Dynamic Solver loading:** On import, OptiLog will automatically bind all its incorporated solvers and the user-provided ones. A user may include a new SAT solver by setting the `OPTILOG_SOLVERS` environment variable to the path where the .so files added by the user are located.

3. **Out of the box SAT solver integration:** OptiLog integrates several state-of-the-art SAT solvers as .so files that can be directly used in Python: Cadical 1.0.2 and Cadical 1.5.2 [31], Glucose 4.1 and Glucose 3.0 [19], Picosat [27], Minisat [45] and Lingeling 18 [29]. Unlike in PySAT these solvers can be also parameterized from OptiLog.

4. **Fast and Flexible Development:** This new way of managing External Libraries allows fast integration of the solvers by decoupling the solvers from the framework itself. Moreover, now solver developers are in control of the compilation pipeline, which allows them to link or include external libraries and modules. On the other hand, PySAT requires solver developers to implement their interface with a full binding through Python's C API and a corresponding Python wrapper class, forcing developers to have extensive knowledge of Python's C API. OptiLog's approach is simpler and less error-prone by removing code repetition and does not require solver developers to have Python knowledge.

   Since the last release, we have also expanded the capabilities of the interface. Now with automatic support for bz2 compression, solver cloning, SIGINT handling, and multithread support:

5. **Signal handling and multithread support:** OptiLog is able to handle SIGINT signals while providing multithread support by releasing the Global Interpreter Lock (GIL) before calling `solve` or `propagate`, while PySAT cannot handle signals when multithread support is enabled. This may be a bottleneck for complex SAT-systems where one thread is executing a SAT solver and

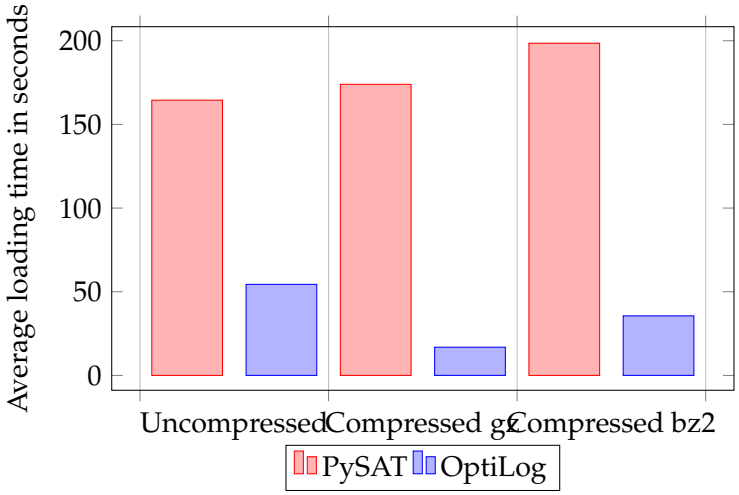
other threads are dedicated to other tasks. In this case, PySAT can not manage signals sent to the SAT solver.

Moreover, OptiLog handles SIGINT signals gracefully, allowing multiple SIGINT signals to be caught and handled, meanwhile, PySAT implementation blocks and does not throw exceptions after the first signal.

6. **Support for Solver cloning:** We have added support for solver cloning. Solver cloning allows solvers to copy their current state and create a new solver object that is equivalent to the original. Cloning can involve replicating the internal state of all the data structures of the solver. The goal is that we can guarantee that the search in the new solver will evolve exactly as in the original solver at the point where the cloning was performed.

7. **Efficient formula loading:** In contrast with PySAT, our formula loading methods are implemented directly in C++, which allows very efficient loading. These formulas may be loaded into Formula objects or directly into the solvers, avoiding the inefficient representation in Python. OptiLog provides support for .cnf and .wcnf files that may be compressed with gz or bz2. Here we can find a comparison in runtime speed between PySAT and OptiLog. The benchmark instantiates a Glucose 4.1 solver and loads the hard clauses of every WCNF instance in the Incomplete Weighted track of the MaxSat Evaluation 2021. The graph below shows the mean of the loading times of the 33 uncompressed instances that took more than 30 seconds to load on PySAT. OptiLog was able to load all the instances with less than 12GB of RAM, while PySAT required 36GB.

   Their average size of the instances in number of clauses is 34.5M, while their average size in MB for uncompressed, compressed in .gz, and in .bz2 is 1055.5 MB, 176.37 MB, and 102.7 MB respectively. Additionally, the largest instance that we used had 132.7M clauses, and a size of 3.7GB, 501MB, 200MB for its uncompressed version, compressed with .gz, and .bz2 respectively.



6.5 BlackBox Module

Executing *External tools* such as: generators, preprocessors, feature extractors, binaries of other solvers, or any other system command, usually becomes a very ad-hoc and contrived procedure. These are the main contributions:

1. **Execution and configuration of External Tools:** We have developed the *BlackBox* module that allows the execution and parsing of arbitrary programs while running in a memory and time-constrained environment with the help of the runsolver tool [94]. These black boxes are also integrated with the *Tuning* module so that they can be automatically configured (see section 6.6). We also provide utilities to parse the output of the execution of these programs with regular expressions.

2. **Integration with Satex:** We have further merged the *Blackbox* module with SAT Heritage [18]. This allows OptiLog to run any SAT solver developed in SAT Competitions [24] since 2000. We are in the process of collecting the configurable parameters for these solvers to enable the tuning of them. We have integrated the wrapper wSatex (see Figure 6.1) that acts as a bridge for the SAT Heritage Docker Images. With this wrapper, OptiLog can call solvers from all the SAT competitions and automatically parse their output. Solvers can be called with .cnf files or by using the CNF formula, which will transparently use memory files that do not write to disk for a more efficient resource usage.

6.6 Tuning Module

This module allows the user to define a configuration scenario to tune a given application (solver, algorithm, function, blackbox, etc). With this scenario OptiLog automatically generates all the files and resources to run independently an automatic configurator (tuner) [3] (see example in the supplementary material). These are the new features we have added:

1. **Automatic deployment of the winning configuration**. We have also simplified the user interaction with the tuning process. Now, the winning configuration reported by the tuner can be automatically recovered. The user can now get a new instance of the application properly set to the wining configuration and ready to be executed.

2. **Automatic configuration of BlackBoxes.** Thanks to the addition of the *BlackBox* module, we now allow to automatically configure any of the executables that the *BlackBox* module can handle (see section 6.5). We follow the same interface that we use to configure Python functions and iSAT solvers, as it can be seen in Figure 6.6. This addition opens the door to use OptiLog to ease the configuration process of any *External tool* and be applied in other research communities, education programs, or industrial environments.

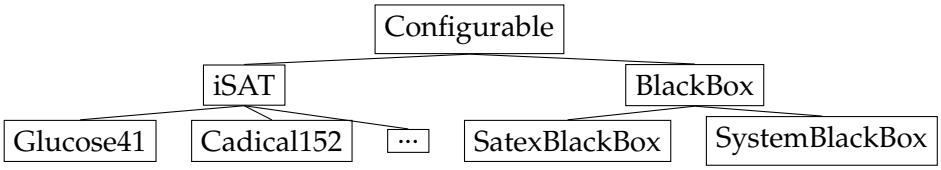


FIGURE 6.6: Diagram of the *Configurable* classes in OptiLog.

As we can see in the example on the Slitherlink problem in the documentation and the supplementary material, by applying the tuner GGA[9] we are able to solve 25% more instances and we decrease the PAR10 metric by 66%.

[3] We currently support GGA [9] and SMAC [75]

6.7 Running Module

To evaluate the performance of an application (e.g. SAT-based system) we typically evaluate it on test data-sets (e.g. set of SAT instances) and analyze the results according to some metric. In this context, a task is the evaluation of the application on a particular data-set.

In this work, we have also extended OptiLog to tackle this tedious and error-prone step by providing an automatic procedure. This procedure submits all these tasks to (potentially) any execution environment, collects and parses the output of these tasks, and aggregates the results. These are the main features of the new *Running* module:

1. **Execution scenario.** An execution scenario is defined by: (i) the application(s), (ii) the data-sets (e.g. SAT instances) that will be used, (iii) the execution limits (CPU, memory...), and (iv) a submitter script.

2. **Running the scenario.** In order to execute the scenario, OptiLog will call a *submitter script* supplied by the user. OptiLog provides example scripts for executing locally (using Task Spooler [76]) and in a High Performance Cluster (using Sun Grid Engine [84]), and potentially on the cloud (currently under development). Note that OptiLog remains agnostic against execution platforms, thus allowing the user to provide ad-hoc scripts for custom execution environments without hassle. Upon execution, the logs for each task will be stored in the scenario directory.

3. **Parsing and aggregating results.** Optilog also provides tools to extract information of the logs, once the experiment has finished. It will read the raw output of each task and will parse the information that the user specifies using *filters*. The information is then presented to the user as a *Pandas* [99] dataframe. We decided to use this data structure as, over the years, it has become the *de facto* standard by data scientists. Its flexibility, as well as the large support from third-party tools (such as visualization tools), allows the user to extract insights from the results of the experiment painlessly.

   A *filter* is defined by:

   - The regular expression for the value to be extracted.
   - If we want to retrieve all the matches of the regular expression.
   - If we want to store the timestamp from when the value was reported.
   - A name for the value. This will correspond to a column in the dataframe.

   In particular, for SAT-based solvers that follow a standardized output format, we provide some templates. Currently, we support SAT solvers and MaxSAT solvers that conform to the DIMACS output format used by the SAT competition [24] and MaxSAT evaluation [22]. For example, the SAT template comes with filters to detect the satisfiability of a formula and its model. The MaxSAT filter also detects the cost.

Competition organizers, research groups, etc. have already their own tools, less or more automatized, to carry out their experiments. This is a very time-consuming part, error-prone of the research process and a non-negligible task of software engineering. Even in the same research group or community different individuals reinvent the wheel systematically when it has to do with managing experiments.

OptiLog aims to provide a common baseline so that we can all build on top and mitigate the impact of this step in the development process. Moreover, having this common base we enforce the reproducibility and trust on experimental results.

Chapter 7

Conclusions and Future Work

In this section, we conclude with the approaches and experimental investigation conducted in this thesis.

From the results in Chapter 3 we conclude that MaxSAT technology is well-suited for solving the Covering Array Number problem for Mixed Covering Arrays with Constraints through SAT technology. In particular, we discussed efficient encodings and how MaxSAT algorithms perform on them.

We also presented MaxSAT encodings for the Tuple Number problem. To our best knowledge, this is the first time that this problem is studied with SUT Constraints. Additionally, we presented a new incomplete algorithm that can be applied efficiently to solve those instances where the Tuple Number problem encoding into MaxSAT is too large. In particular, we proved we can build good enough solutions by incrementally adding a new test synthesized through a MaxSAT query that aims to maximize the coverage of additional allowed tuples, with respect to the test suite under construction.

Another interesting result that we obtained is that if we do not aim to cover all t-tuples but a *statistically significant* fraction, we can save a great number of tests. We experimentally showed that to cover a 95% percentage, we just need, on average, a 52% percentage of the best suboptimal solution reported so far. This is of high practical importance for applications where test cases are expensive according to the budget.

From the point of view of Combinatorial Testing, it is reasonable to say that the practical and theoretical interest application of our findings and approaches will grow proportionally to the hardness or complexity of the SUT constraints. This will certainly extend the reach of Combinatorial Testing to more challenging SUTs.

From the point of view of Constraint programming, the lessons learnt on how to design efficient encodings for MaxSAT solvers can be exported to solve similar problems. These problems are roughly characterized by having an objective function whose size is proportional to the best-known upper bound.

SAT and MaxSAT communities will also benefit from new challenging benchmarks to test the new advances in the field. Moreover, any future advance in MaxSAT technology can be applied to solve more efficiently the Covering Array Number and Tuple Number problems with no additional cost.

From the results in Chapter 4, we conclude that 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. Thanks to this low memory consumption plus the partitioning nature of the Pool based approach, we have also presented a parallel version of our algorithms that provides a practical approach to be applied in real environments. Our experimental results provide improved upper bounds for 16 benchmarks.

From the results in Chapter 5, we conclude that thanks to the original and simple design of our new generator for SUT benchmarks, now it is possible to have access, from the CT research community to many different kinds of structures contained in SAT instances that do represent real-world problems. In particular, we can now generate SUT benchmarks of a given size and constraint hardness that will ease the development of new CT tools. Additionally, this approach can be easily adapted to take advantage of the available instances in other constraint programming or operation research formalisms. Finally, an important contribution of this work is to have provided a first detailed study of available CT tools. This study helps to characterize better when we should use a given CT tool and therefore increase the robustness of our testing strategies when facing a new SUT.

As future work, about SAT-based Combinatorial Testing approaches, we will continue exploring how to better integrate Satisfiability and other Constraint Programming technology into our algorithms, explore other applications of Covering Arrays and come up with variations and extensions that can better satisfy the industrial requirements. Moreover, we plan to extend *CTLog* to become the CT tool of reference.

Finally, from Chapter 6 we conclude that the new extensions for the framework OptiLog open a new range of applications. These new applications range from supporting researchers, educators, and practitioners to create more ambitious end-to-end applications cases where SAT plays a key role. OptiLog allows focusing our energy on modelling and solving problem issues while still being able to carry out comprehensive experimental studies involving also tuning steps.

As future work, we plan to enrich the OptiLog framework. From a more technical perspective, we plan to extend it with support for distributed algorithms (including cloud computing) and a new module for metaheuristic algorithms. From a more educational point of view, we will generate a database of course assignments with instructions for educators and students including autograders.

Bibliography

[1] Amir Aavani. Translating pseudo-boolean constraints into cnf. In Karem A. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, pages 357–359, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Josep Alos, Carlos Ansótegui, Josep M. Salvia, and Eduard Torres. Optilog V2: model, solve, tune and run. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 25:1–25:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[3] Mario Alviano, Carmine Dodaro, and Francesco Ricca. A maxsat algorithm using cardinality constraints of bounded size. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[4] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196:77–105, 2013.

[5] Carlos Ansótegui, Idelfonso Izquierdo, Felip Manyà, and José Torres-Jiménez. 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*, pages 51–59, 2013.

[6] 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.

[7] 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.

[8] Carlos Ansótegui, Jesus Ojeda, and Eduard Torres. Building high strength mixed covering arrays with constraints. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 12:1–12:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[9] Carlos Ansótegui, Josep Pon, and Meinolf Sellmann. Boosting evolutionary algorithm configuration. *Annals of Mathematics and Artificial Intelligence*, 2021.

[10] Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy. Improving SAT-Based Weighted MaxSAT Solvers. In Michela Milano, editor, *Principles*

and Practice of Constraint Programming, Lecture Notes in Computer Science, pages 86–101, Berlin, Heidelberg, 2012. Springer.

[11] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 427–440, Berlin, Heidelberg, 2009. Springer.

[12] Carlos Ansótegui and Joel Gabàs. WPM3: An (in)complete algorithm for weighted partial MaxSAT. *Artificial Intelligence*, 250:37–57, September 2017.

[13] Carlos Ansótegui, Joel Gabàs, and Jordi Levy. Exploiting subproblem optimization in SAT-based MaxSAT algorithms. *Journal of Heuristics*, 22(1):1–53, February 2016.

[14] 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.

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

[16] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. Improving glucose for incremental sat solving with assumptions: Application to mus extraction. In *International conference on theory and applications of satisfiability testing*, pages 309–317. Springer, 2013.

[17] Gilles Audemard, Loïc Paulevé, and Laurent Simon. SAT heritage: A community-driven effort for archiving, building and running more than thousand SAT solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 107–113. Springer, 2020.

[18] Gilles Audemard, Loïc Paulevé, and Laurent Simon. Sat heritage: A community-driven effort for archiving, building and running more than thousand sat solvers. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 107–113, Cham, 2020. Springer International Publishing.

[19] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[20] Florent Avellaneda. A short description of the solver evalmaxsat. *MaxSAT Evaluation 2020*, page 8.

[21] Fahiem Bacchus. Maxhs in the 2020 maxsat evaluation. *MaxSAT Evaluation 2020*, page 19.

[22] Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.

[23] Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Rubens Martins. Maxsat evaluation 2020: Solver and benchmark descriptions. 2020.

[24] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, Finland, 2021.

[25] 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.

[26] Jeremias Berg, Emir Demirovic, and Peter Stuckey. Loandra in the 2020 maxsat evaluation. *MaxSAT Evaluation 2020*, page 10.

[27] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

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

[29] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT competition*, 2013:1, 2013.

[30] 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.

[31] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[32] 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.

[33] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[34] 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.

[35] 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.

[36] Oscar Carrizales-Turrubiates, Nelson Rangel-Valdez, and José Torres-Jiménez. Optimal shortening of covering arrays. In Ildar Z. Batyrshin and Grigori Sidorov, editors, *Advances in Artificial Intelligence - 10th Mexican International Conference on Artificial Intelligence, MICAI 2011, Puebla, Mexico, November 26 - December 4, 2011, Proceedings, Part I*, volume 7094 of *Lecture Notes in Computer Science*, pages 198–209. Springer, 2011.

[37] Francois Chollet et al. Keras, 2015.

[38] 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.

[39] COIN-OR Foundation. Computational infrastructure for operations research. https://www.coin-or.org/, 2016.

[40] Charles J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, 59(1,2):125–172, 2004.

[41] 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.

[42] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[43] dimacs.rutgers.edu. Dimacs cnf suggested format, 2021.

[44] 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.

[45] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[46] 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.

[47] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, pages 462–476, 2002.

[48] Yingjie Fu, Zhendong Lei, Shaowei Cai, Jinkun Lin, and Haoran Wang. Wca: A weighting local search for constrained combinatorial test optimization. *Information and Software Technology*, 122:106288, 2020.

[49] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[50] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Sat solving for termination analysis with polynomial interpretations. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 340–354, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[51] Python Software Fundation. Python package index - pypi.

[52] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, Gregor Hendel, Christopher Hojny, Thorsten Koch, Pierre Le Bodic, Stephen J. Maher, Frederic Matter, Matthias Miltenberger, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Christine Tawfik, Stefan Vigerske, Fabian Wegscheider, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 7.0. ZIB-Report 20-10, Zuse Institute Berlin, March 2020.

[53] Angelo Gargantini and Marco Radavelli. Migrating combinatorial interaction test modeling and generation to the web. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 308–317, 2018.

[54] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.

[55] Brady J. Garvin, Myra B. Cohen, and Matthew B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *2009 1st International Symposium on Search Based Software Engineering*, pages 13–22, 2009.

[56] 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.

[57] Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 274–289, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[58] Google. Google OR-Tools. https://developers.google.com/optimization, 2021.

[59] 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.

[60] T. Guns. Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example. In *The 18th workshop on Constraint Modelling and Reformulation at CP (ModRef 2019)*, 2019.

[61] Gurobi Optimization. Gurobi. https://www.gurobi.com/, 2021.

[62] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[63] 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.

[64] 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.

[65] 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.

[66] IBM. IBM ILOG CPLEX: High-performance software for mathematical programming and optimization, 2006.

[67] Alexey Ignatiev. Rc2-2018@ maxsat evaluation 2020. *MaxSAT Evaluation 2020*, page 13.

[68] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[69] Serdar Kadioglu. Column generation for interaction coverage in combinatorial software testing, 2017.

[70] 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.

[71] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017.

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

[73] Yu Lei, Raghu Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[74] Zhendong Lei and Shaowei Cai. Solving (weighted) partial maxsat by dynamic local search for sat. In *Proceedings of the Twenty-Seventh International*

Joint Conference on Artificial Intelligence, IJCAI-18, pages 1346–1352. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[75] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization, 2021.

[76] Lluis Batlle i Rossell. Task spooler man page, 2021.

[77] Logic and Optimization Group. OptiLog C++ Interface for Python bindings. https://github.com/optilog/OptiLogFrameworkInterface. Accessed: 2022-02-26.

[78] Logic and Optimization Group. Pypblib. https://pypi.org/project/pypblib/, 2019. University of Lleida.

[79] Logic and Optimization Group. OptiLog official documentation, 2022.

[80] Logic Optimization Group. PyPBLib: PBLib Python3 bindings. https://pypi.org/project/pypblib/, 2018. Described in OptiLog [7].

[81] Chuan Luo, Jinkun Lin, Shaowei Cai, Xin Chen, Bing He, Bo Qiao, Pu Zhao, Qingwei Lin, Hongyu Zhang, Wei Wu, Saravanakumar Rajmohan, and Dongmei Zhang. Autoccag: An automated approach to constrained covering array generation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 201–212, 2021.

[82] 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.

[83] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for Weighted Boolean Optimization. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, Lecture Notes in Computer Science, pages 495–508, Berlin, Heidelberg, 2009. Springer.

[84] W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, page 35, USA, 2001. IEEE Computer Society.

[85] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided maxsat solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013.

[86] Alexander Nadel. Tt-open-wbo-inc-20: an anytime maxsat solver entering mse'20. *MaxSAT Evaluation 2020*, page 32.

[87] 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.

[88] Toru Nanba, Tatsuhiro Tsuchiya, and Tohru Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE Transactions*, 95-A(9):1501–1505, 2012.

[89] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

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

[91] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[92] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[93] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[94] Olivier Roussel. Controlling a solver execution: the runsolver tool. *JSAT*, 7:139–144, 11 2011.

[95] SAT-Competition, 2009. www.satcompetition.org/2009.

[96] 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.

[97] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[98] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[99] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[100] 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.

[101] 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.

[102] 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.

[103] Linbin Yu, Feng Duan, Yu Lei, Raghu N. Kacker, and D. Richard 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.

[104] Linbin Yu, Yu Lei, Mehra Nourozborazjany, Raghu N. Kacker, and D. Richard Kuhn. An efficient algorithm for constraint handling in combinatorial test generation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 242–251, 2013.