# Conflict Driven Cylindrical Algebraic Coverings for Nonlinear Arithmetic in SMT Solving

Hanna Franzen

Master Thesis

13th February 2020

Examiners:
Prof. Dr. Erika Ábrahám,
Prof. Dr. Jürgen Giesl

Advisor:
Gereon Kremer M.Sc.

RWTH Aachen University
Informatik 2: Theory of Hybrid Systems

## Abstract

The Cylindrical Algebraic Decomposition (CAD) algorithm is an established algorithm which can be used to decide the Satisfiability (SAT) of problems from the theory of non-linear real arithmetic. Based on the same ideas, the new Cylindrical Algebraic Covering (CAC) algorithm based on using coverings instead of decompositions was recently introduced [33]. It is designed to be used in SAT Modulo Theories (SMT) solving. The CAC algorithm was implemented as the main work for this thesis. It was realized as a module for the toolbox SMT-RAT. The mechanisms used in the implementation and the results of testing the implementation against an implementation of the CAD algorithm are presented in this thesis.

# Contents

**Contents** **v**

**1 Introduction** **1**

**2 Related Work** **3**
2.1 Satisfiability Modulo Theories Solving . . . . . . . . . . . . 4
2.2 Cylindrical Algebraic Decomposition . . . . . . . . . . . . . 5

**3 Theoretical Background** **9**
3.1 Satisfiability Modulo Theories solving . . . . . . . . . . . . 9
3.2 Cells . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

**4 Algorithm: Finding Cylindrical Algebraic Coverings** **13**

**5 Implementation** **21**
5.1 Non-Incremental Approach . . . . . . . . . . . . . . . . . . . 22
5.2 Incremental Approach . . . . . . . . . . . . . . . . . . . . . 34

**6 Test results** **37**

**7 Conclusion** **41**

**Glossary** **43**

**Bibliography** **45**

# 1 Introduction

The Satisfiability (SAT) decision problem is one of the most fundamental problems in computer science. It describes the problem to decide the satisfiability of first-order Boolean formulae. A number of more complex problems can be abstracted to the SAT problem.

One of these problems is the SAT Modulo Theories (SMT) decision problem where the formula is not limited to Boolean variables but is composed of constraints from a certain theory. Hence, the variables can have values in an arbitrary domain and there can be additional operations and relations, accordingly. The topic of SMT solving is an often researched problem but there are some basic mechanics that are always present. The input of an SMT solver is a formula consisting of a Boolean combination of constraints that is to be examined. The SMT solver outputs a Boolean value that describes the satisfiability of the input formula. Additionally, the output includes a satisfying variable assignment or a reason for unsatisfiability, respectively.

The basic SMT solver is usually a combined system of solvers for two domains, a SAT solver and a theory solver. The SAT solver examines the Boolean structure of the input formula and can compute subsets of constraints such that the satisfiability of their conjunction implies the satisfiability of the input formula. The theory solver takes into account the underlying logic, called theory in this context, to calculate the satisfiability of the conjunction of the constraints from the subset.

In this thesis, a new algorithm designed to serve as a theory solver is explored and implemented. The algorithm used is the Cylindrical Algebraic Covering (CAC) algorithm recently introduced by Ábrahám, Davenport, England, and Kremer [33]. It works on the theory of non-linear real arithmetic (NRA).

The new CAC algorithm is based on the established Cylindrical Algebraic Decomposition (CAD) algorithm that was first introduced by Collins in 1975 [16]. The CAD algorithm is a method for quantifier elimination for the real closed field, a problem that was shown to be decidable by Tarski [31]. Relatively to Tarski's method, the CAD method is much more ef-

ficient and was the first method to be practical [24]. As Tarski noted, a quantifier elimination procedure can be used as a decision method [16]. Hence, the CAD algorithm is used in SMT solving to decide the satisfiability of constraints over the theory of non-linear real arithmetic. The CAC algorithm aims at being faster than the CAD algorithm for the computation of unsatisfiability.

A history of SMT solvers and their usage as well as an overview of the CAD algorithm are provided in Chapter 2. The theoretical background of the basic concepts used in this thesis is introduced in Chapter 3. In Section 3.1 the process of SMT solving is explained in detail. The concepts on which the CAD algorithm is based are introduced in Section 3.2 as a foundation for understanding the CAC algorithm. The basic process of the CAC algorithm is explored in Chapter 4. The actual implementation and its mechanisms are described in Chapter 5. In Chapter 6, the results of testing the implementation are presented. The overall results of the thesis are concluded in Chapter 6.

# 2 Related Work

One of the basic problems in the field of theoretical computer science is the *Boolean Satisfiability (SAT)* decision problem. The problem is to decide whether a formula in propositional (Boolean) logic is satisfiable, i.e. whether there exists an assignment to the Boolean variables in the formula such that the formula resolves to true. There was and still is a lot of research into the topic. Modern SAT solver implementations are capable of handling problems with millions of constraints and hundreds of thousands of variables [28]. SAT solving can be used to solve low level abstraction of a lot of practical problems from a wide range of fields like, for example, verification, image computation, and scheduling [21, 23]. The implementation that is presented in this thesis is used in the context of SAT Modulo Theories (SMT) solving, which is a process that uses a SAT solver as one of its main components. Details on the background of SMT solving are given in Section 2.1.

This thesis is written with regard to the 2020 paper on deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using *cylindrical algebraic coverings* by Ábrahám, Davenport, England, and Kremer [33]. The paper describes a new algorithm to solve the satisfiability of conjunctions of non-linear real arithmetic constraints in the context of SMT solving. The algorithm will be referred to as the *Cylindrical Algebraic Covering (CAC) algorithm* in this thesis. As the implementation of the CAC algorithm is the main work of this thesis, it will be described in detail in Chapter 4. The algorithm is based on the idea of the *Cylindrical Algebraic Decomposition (CAD)* algorithm that was first described by Collins in 1975 [16]. Details on the history and variants of the CAD algorithm are introduced in Section 2.2. Due to the suggested application of the CAC algorithm as part of an SMT solver, some background knowledge on such solvers is introduced in the following section.

## 2.1 Satisfiability Modulo Theories Solving

The basic structure of an *SMT solver* usually consists of a SAT solver and a theory solver. The input to such solvers are formulae in a certain theory for which a theory solver is available. A formula is a Boolean combination of constraints. The SAT solver examines the Boolean structure of the formula by viewing the constraints as Boolean variables. The theory solver solves a partial problem determined by the SAT solver consisting of a set of constraints that the SAT solver determined to be true in a valid Boolean assignment and the negation of the other constraints. If the theory solver determines the conjunction of the constraints in this set to be satisfiable, the formula can be satisfied. Else, there is a conflict and the theory solver gives an unsatisfiable subset of the constraints to the SAT solver as an explanation. The SAT solver can add a conflict clause to the original formula and backtrack to try to find another Boolean-wise valid assignment. This process in the SAT solver is based on the Davis–Putnam–Logemann–Loveland (DPLL) procedure and is called conflict-driven clause learning (CDCL). The details of the SMT solving process will be introduced in Chapter 3. Over the years, optimizations to and new procedures based on this basic process have been proposed.

A popular SMT approach was developed by de Moura and Jovanović in 2013 [19]. The technique is called model-constructing satisfiability calculus (mcSAT) and is an adaption of the DPLL procedure. In mcSAT, the conflict resolution is not restricted to Boolean decisions in the SAT solver. In this approach, the SAT solver includes the building of a model in the given theory. The theory solver is used when searching for a conflict once the SAT solver found a satisfying Boolean solution. The creation of new literals is allowed to describe the conflict in the formula and values can be concreted by variable assignments.

There are several implementations of SMT solvers. A lot of them are regularly updated according to the current research. Their development can, for example, be seen in the International Satisfiability Modulo Theories Competition held every year [10].

One of the most popular SMT solver implementations is Z3 which was developed by de Moura and Bjørner as part of Microsoft Research [18]. Z3 does not only consist of the obligatory SAT and theory solvers but also includes a number of simpifiers and (pre-)processor modules. The Z3 solver was applied to a number of real-world problems, including extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains [18], but also applications less close to computer science theory like in biological computations [32]. The Z3 tool and the corresponding paper have won several awards. To name

some, they won the Test of Time Award in 2018 [30] for the often cited 2008 paper and the Herbrand Award for Distinguished Contributions to Automated Reasoning in 2019 for the overall contribution to SMT solving [29]. Z3 is open source software, its code is available on GitHub [4].

Another open source SMT solver is Yices [3]. It is developed by the SRI International's Computer Science Laboratory [6]. It is capable of solving a range of first-order theory problems applicable to solve problems concerning, for example arithmetic, uninterpreted functions, bit vectors, arrays, and recursive data types [20]. Its four main theory solvers are specialized on uninterpreted functions with equalities, linear arithmetic, bitvectors, and arrays. In 2019, the Yices 2.6.2 incremental solver won the first place for solving linear integer arithmetic in the 14th International Satisfiability Modulo Theories Competition [8].

Another SMT solver is SMT-RAT. It is a toolbox created as a modular solver in which a custom strategy decides which theory solvers are used to solve the input problem. The toolbox is developed at the RWTH Aachen University in the group of Theory of Hybrid Systems, [17]. SMT-RAT is the SMT solver that is used in this thesis. With its default configuration, it has won the first place in the 13th International Satisfiability Modulo Theories Competition for quantifier-free non-linear mixed integer real arithmetic (QFNIRA) in 2018 and 2019 [7, 9]. SMT-RAT's code is open source and published on GitHub [2]. While the CAC algorithm as described in [33] is meant for dealing with real numbers, the realization of such is difficult on computers. SMT-RAT actually calculates on algebraic numbers. The underlying arithmetic is computed using the CArL library that is developed along with SMT-RAT [1]. To handle problems on real numbers, the toolbox provides a wrapper class for algebraic numbers that can be treated similarly to real numbers by the programmer.

There is an international initiative to unify input and output languages of SMT solvers and provide a strong community for the research topic of SMT solving. It is organized in the Satisfiability Modulo Theories Library (SMT-LIB) [5] and also provides a large collection of benchmark units over different logical theories.

## 2.2 Cylindrical Algebraic Decomposition

The original *CAD algorithm* was developed in the 1970s by Collins [16]. The context for the algorithm was not SMT solving but it was originally inspired by Tarski's quantifier elimination procedure [24]. The CAD procedure was described as an algorithm that "given a set of $r$-variate integral polynomials, a cylindrical algebraic decomposition of euclidean $r$-

space $E^r$ partitions $E^r$ into connected subsets compatible with the zeros of the polynomials" is calculated [11, p.1]. The expected theory of the input formulas is stated to be well-formed in the first-order theory of the real closed field [11], just like Tarski's procedure used the theory of the real closed field [24]. The usage of the CAD algorithm to eliminate quantifiers is adaptable to the SMT context because the formula can be treated as being fully existentially quantified. The idea for the CAC algorithm examined in this paper is based on the established CAD algorithm.

The basic CAD algorithm has a *projection* and a *lifting* phase [12]. In the projection phase, the algorithm eliminates a variable from the constraints' polynomials. In order to preserve the original sign-invariant regions of the polynomial set, new polynomials can be added during this process. Sign-invariant regions are regions in which none of the polynomials change their algebraic sign. This is done consecutively for each but one variable. How exactly this projection works depends on the used projection operator. The original one proposed by Collins [16] was improved many times, e.g. by McCallum in 1985 [26], by Hong in 1990 [22] or, recently proven to be valid, by Lazard in 1994 [25, 27].

After the projection is finished, the lifting phase begins where the one-dimensional domain is decomposed into sign-invariant regions for the remaining variable. This is the one-dimensional CAD. Then, consecutively, the polynomials from the next higher dimension in the projection are filled with samples from the determined regions for the lower dimensional variables for each cell. This results in as many possible variants of the polynomials as there were cells in the last dimension's CAD. These are, again, divided into sign-invariant regions to be the next dimension's CAD. This is repeated until the full-dimensional CAD is determined. To put it differently, a CAD is a partitioning of the state space into connected subsets which are called *cells*. To decide satisfiability, one can determine whether one of the CAD's cells is satisfying. Then the whole problem is satisfiable by assigning values from the cell to the variables.

Over the years, a lot of simplifications and adaptations of the CAD algorithm were proposed and implemented. One branch of approaches, for example, are the single (open) cell approaches by Brown and Košta [13, 15]. They take a single point and a set of polynomials as input and construct a cell around this point that is broader than the original CAD cell would be. Partial approaches like this one can be embedded in other adaptations of the CAD algorithm. The mentioned approaches by Brown and Košta were inspired by a new, also CAD-based approach for solving non-linear arithmetic by Jovanović and de Moura called NLSAT [24]. As the CAD algorithm computes new conflict polynomials based on the original polynomials, the problem size can explode. To put it

briefly, Jovanović and de Moura try to reduce this size by backtracking and involving the model, i.e. the assignment to the variables, that the algorithm tried to build into the conflict resolution. Using this additional information, the algorithm aims at determining only a conflict core instead of all possible conflict polynomials.

An approach based on Jovanović's and de Moura's NLSAT algorithm is the creation of an open Non-uniform Cylindrical Algebraic Decomposition (NuCAD) as introduced by Brown in 2015 [14]. In this approach, cells still have to be cylindrical but the necessity of being cylindrically arranged with respect to the other cells in the decomposition is omitted. This results into a decomposition with fewer cells than the regular CAD. The approach by Brown does not have to build the full projection at the beginning but can operate iteratively. Using the model-based approach by Jovanović and de Moura [24] and the iterative proceeding, Brown's approach has an easier computation for the projection onto lower dimensions than the basic CAD algorithm.

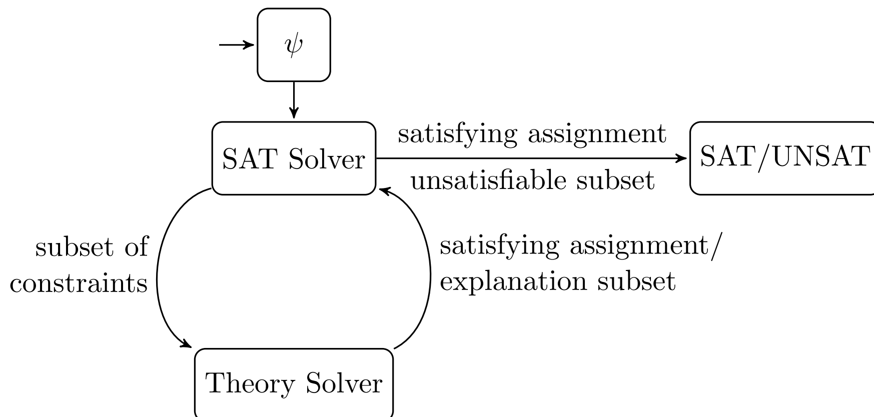
# 3 Theoretical Background

## 3.1 Satisfiability Modulo Theories solving

*Satisfiability (SAT) Modulo Theories (SMT) solving* usually is the combination of a *SAT solver* and one or more *theory solvers*. In this thesis the concept of a Cylindrical Algebraic Covering (CAC) algorithm will be used to solve theory problems in the context of SMT solving. The algorithm that is being examined can be used in an SMT solver as a theory solver.

For an overview of the SMT solving process, have a look at figure 3.1 on page 10. The input of an SMT solver is a quantifier-free first-order logic formula, i.e. a Boolean combination of constraints over an arbitrary theory for which a theory solver is provided. In the figure the input formula is depicted as $\psi$. The solver actually uses the set of all constraints from the formula and their negations. In general, the SAT solver regards the constraints as Boolean variables as an abstraction and tries to find a satisfying assignment for the Boolean formula. If the conjunction of this subset of Boolean-wise true constraints and the negation of the other constraints is satisfiable, the input formula can be satisfied. All of these constraints that the SAT solver marked as true and the negation of the other ones are given to the theory solver. The theory solver can then determine whether this subset of the known constraints is satisfiable. For this thesis the underlying theory is assumed to be quantifier-free non-linear real arithmetic (QFNRA). If the theory solver decides that there is a satisfying assignment it gives such a solution to the SAT solver. The SAT solver returns the assignment, possibly in combination with its own results. In case the theory solver finds that the given conjunction is unsatisfiable it should give the SAT solver an explanation, e.g. a subset of the constraints that contains the contradiction. From this explanation a new term called a conflict term can be build that is added to the original formula. Again, the SAT solver checks the Boolean formula, but due to the conflict term excludes the former theory conflict. The new subset of constraints can again be passed to the theory solver until either the theory solver can find a satisfying assignment in respect to the given subset of constraints or the

Figure 3.1: Schematics of an SMT solver




SAT solver determines the Boolean abstraction of the formula including the conflict terms to be unsatisfiable.

There are two basic concepts of how the theory solver deals with sets of constraints passed by the SAT solver. Either the theory solver checks the whole subset of constraints for unsatisfiability. This is called non-incremental theory solving. Alternatively, the theory solver stores the previous calculations and works only on the changes of the subset, i.e. extends or reverses the former calculation only regarding newly added or removed constraints. This kind of theory solvers is iterative, working incrementally. In this case the SAT checker may only pass the changes and the theory solver has the capability to backtrack and apply changes to its former data according to the changed constraints.

To understand how the algorithm that is examined in this thesis works, a more precise definition of a constraint is necessary. To introduce the notion of constraints as used in this thesis, a definition of polynomials is needed.

**Definition** (Polynomial)**.** *A* polynomial *in $n$ variables is defined as*

$$p(x_1, \ldots, x_n) := \sum_{i=1}^{m} c_i \cdot \prod_{k=1}^{n} x_k^{e_{ik}}$$

*where $m$ is an arbitrary positive integer, $n$ is the number of variables, and $e_{ik}$ is the positive integer or zero exponent of the variable $x_k$ in the product $i$. The constants $c_i$ are limited to rational numbers for this thesis. The terms* coefficient *and* leading coefficient *are used as usual.*

**Definition** (Constraint). *A* constraint *consists of a polynomial in relation to zero, so a constraint c is defined as*

$$c := p(x_1, \ldots, x_n) \sim 0, \qquad \text{where} \sim \in \{<, >, \leq, \geq, =, \neq\}$$

*and $p(x_1, \ldots, x_n)$ is an arbitrary polynomial in variables $x_1, \ldots, x_n$.*

The domain of the polynomials is $\mathbb{R}^n$, so the underlying theory is the non-linear real arithmetic (NRA). The input is expected not to contain quantifiers, so NRA can be further restricted to QFNRA. The input formula $\psi$ as shown in Figure 3.1 is a Boolean combination of constraints. The input of the theory solver is a subset of the constraints contained in the formula or their negations. This subset is implicitly treated as a conjunction of the constraints.

## 3.2 Cells

To understand how the CAC algorithm works, the concept of *cells* as used in the Cylindrical Algebraic Decomposition (CAD) algorithm is introduced in its most common form [11, 33]. Note that in some literature cells are called *regions* [11] instead. The term region is used differently in this thesis and will be introduced later in this chapter.

**Definition** (Cell). *A* cell *is a non-empty connected subset of $\mathbb{R}^n$.*

In the CAD context, only algebraic cells are considered. As the algorithm examined in this thesis is based on the ideas of the CAD algorithm, this characteristic should be established.

**Definition** (Algebraic Cells). *A cell is* algebraic *if and only if it can be defined by a conjunction of algebraic constraints.*

For example, an algebraic cell in $\mathbb{R}^1$ could be defined by the conjunction $x_0 + 2 > 0 \land x_0 < 0$.

A CAD is a *cylindrical decomposition*, so this term must be introduced, too.

**Definition** (Decomposition). *A* decomposition *$D$ is a partitioning of $\mathbb{R}^n$ into disjoint cells:*

$$D \subseteq C \text{ such that } \bigcup_{c \in D} = \mathbb{R}^n \text{ and } \forall c_i, c_j \in D.\ c_i \cap c_j = \emptyset$$

*where $C$ is the set of all cells in $\mathbb{R}^n$.*

**Definition** (Cylindrical decomposition)**.** *A decomposition is* cylindrical *if and only if the arrangement of its cells is cylindrical. This is the case if for each pair of cells in the decomposition of dimension* $\mathbb{R}^i$*, the projections of these cells onto the dimension* $\mathbb{R}^{i-1}$ *are either identical or disjoint.*

In the basic CAD algorithm, cells are sign-invariant for the polynomial of each constraint in every dimension. This will not necessarily be the case in the algorithm presented in Chapter 4, but sign-invariance is still the key idea to build the current dimension's cell part. The definition is given as in the CAC source paper [33].

**Definition** (Sign-Invariance)**.** *A cell* $C$ *is sign-invariant for a polynomial* $p$ *if and only if exactly one of the following conditions hold*
$\forall\, x \in C.\quad p(x) < 0,$
$\forall\, x \in C.\quad p(x) > 0,$
$\forall\, x \in C.\quad p(x) = 0.$

More verbose, this means that a cell is sign-invariant for a polynomial if in the whole cell the polynomial evaluates to values that are solely above or below zero, or evaluates to exactly zero for the whole cell. A cell is sign-invariant for a constraint if it is sign-invariant for the constraint's polynomial. As the constraints relate the polynomials to zero one can easily determine whether the constraint is valid in a sign-invariant cell. This can also be applied to sub-cells of current dimension $i$ with variable $x_i$ if the evaluated constraint does not actually use the variables $x_{i+1}, \ldots, x_n$ of higher dimensions.

The CAC algorithm aims to find cells in which the unsatisfying sign-invariant regions of the current dimension cover $\mathbb{R}$. Note that in this context, the term "unsatisfying sign-invariant region of the current dimension" refers to an unsatisfying interval in the current dimension. Intervals are defined as usual with a lower and an upper bound which can be open or closed. Both bounds are allowed to be unbounded towards the corresponding infinity. The algorithm will see the former dimensions as fixed and only determine such intervals for the current variable $x_i$.

**Definition** (Covering)**.** *A covering* $D$ *of* $\mathbb{R}$ *is defined as a set of intervals such that their union covers* $\mathbb{R}$*, i.e. let* $\mathbb{I}$ *be the set of intervals in* $\mathbb{R}$*, then*

$$D \subseteq \mathbb{I} \text{ such that } \bigcup_{i \in D} = \mathbb{R}.$$

If there is a covering of $\mathbb{R}$ in these intervals for $x_i$, the algorithm will either exclude an appropriate interval from the former dimension, or, if the dimension is the first one, will determine the conjunction of the given constraints to be unsatisfiable.

# 4 Algorithm: Finding Cylindrical Algebraic Coverings

As it is important to understand the details of the Cylindrical Algebraic Covering (CAC) approach to understand the implementation, the CAC algorithm will be described in this chapter. It stems from the paper by Ábrahám, Davenport, England, and Kremer [33]. It is designed to be used as the theory solver of an Satisfiability (SAT) Modulo Theories (SMT) solver. As such, one can assume that there is a set of constraints that was passed by the SAT solver according to the mechanisms introduced in Chapter 3. Hence in the context of SMT solving, the objective of the CAC algorithm is to determine whether the conjunction of the given constraints is satisfiable. The output data should also include a subset of these constraints whose conjunction is unsatisfiable, if the result is unsatisfiability. In case of satisfiability of the conjunction, the result should include a satisfying witness, i.e. a satisfying set of assignments of values to the variables.

Note that the presented CAC algorithm is not exclusively useful for the purpose as a theory solver. It can, with little adaptation, be applied to any problem in which a conjunction of constraints is to be solved. It is, however, optimized for the application in an SMT solver and was designed with the search for unsatisfiability in mind. As such, the aim of this thesis is to determine whether the new CAC algorithm is faster than the traditional Cylindrical Algebraic Decomposition (CAD) algorithm with regard to finding a problem to be unsatisfiable. Nonetheless, the algorithm should still find assignments in case of satisfiability, so another point of analysis is the algorithm's performance in satisfiable cases.

The main algorithm to find coverings is based on identifying unsatisfying intervals in a recursive fashion over the variables. Due to the recursion, the algorithm follows a depth-first approach in creating cells. A pseudo

**Algorithm 1:** Main algorithm: get_unsat_cover [33]

**Data:** Set of global constraints defined over $\mathbb{R}^n$, variables $x_1, \dots, x_n$.
**Input :** $s = (x_1 = v_1, \dots, x_{i-1} = v_{i-1})$
**Output:** SAT or UNSAT,
satisfying witness or unsatisfiable interval covering

```
1  𝕀 := get_unsat_intervals(s, x_i)
2  while ⋃_{I∈𝕀} I ≠ ℝ do
3      s_i := sample_outside(𝕀)
4      if i = n then
5          return (SAT, (s_1, ..., s_{i-1}, s_i))
6      (f, O) := get_unsat_cover((s_1, ..., s_{i-1}, s_i))
7      if f = SAT then
8          return (SAT, O)
9      else if f = UNSAT then
10         R = construct_characterization((s_1, ..., s_{i-1}, s_i), O)
11         I = interval_from_
            characterization((s_1, ..., s_{i-1}), s_i, R)
12         𝕀 = 𝕀 ∪ {I}
13 return (UNSAT, 𝕀)
```

code schematic is depicted in Algorithm 1 which was taken from Ábrahám et al. [33].

Briefly, if there are unexplored regions of the domain, a sample for the current variable is taken from these regions and checked for compatibility with the remaining dimensions. If this sample is satisfying and further satisfying samples could be found for the remaining variables, a satisfying witness was found and there is no unsatisfying covering. Else, the reason for the unsatisfiability is explored and a new unsatisfying interval can be added. In this case, the loop restarts and samples from the remaining unexplored regions. The recursion stops once a full sample set for all variables was computed or the whole real domain is covered with unsatisfying intervals.

It is assumed that the variables have a given order and that the samples for the variables are to be computed in that order. In the following, the recursion depth will be referred to as the current dimension $i$. This correlates with the current variable $x_i$ for which the algorithm tries to find a satisfying sample in dimension $i$.

For this algorithm, the notion of intervals is expanded. They include

**Algorithm 2:** get_unsat_intervals [33]

```
   Input  : Sample set of current dimension
   Output: Set of unsatisfiable intervals
1  𝕀 = ∅
2  C_i := constraints with main variable x_i
3  foreach c ∈ C_i do
4      c' := c(s)                                          // c' = p ~ 0
5      if c' = false then
6          return {(−∞, ∞, ∅, ∅, {p}, ∅)}
7      if c' = true then
8          continue
                             // c' is univariate in the i^th variable
9      Z = real_roots(p, s)    // Z = {z_1 ... z_k} ordered ascend.
10     Regions = {(−∞, z_1), [z_1, z_1], (z_1, z_2), ..., (z_k, ∞)}
11     foreach I ∈ Regions do
12         let r ∈ I if c'(r) = false then
13             L, U := ∅
14             if ℓ ≠ −∞ then L := {p}
15             if u ≠ ∞ then U := {p}
16             𝕀 := 𝕀 ∪ {(ℓ, u, L, U, {p}, ∅)}
17 return 𝕀
```

lower and upper bounds as usual, referred to as lower bound $l$ and upper bound $u$. Additionally, they hold a set of polynomials for each bound that describe the bound in the current dimension $i$. These sets of polynomials are referred to as $L$ for the lower bound and $U$ for the upper bound. Further additions are two more sets of polynomials, $P_i$ and $P_\perp$. $P_i$ holds all polynomials with main variable $x_i$ from which the interval originates. $P_\perp$ holds all polynomials with main variables of lower dimensions from which the interval originates.

In detail, first the set of unsatisfying intervals is computed with respect to the former dimensions as described in Algorithm 1 in Line 1. A detailed pseudo code overview of this process is depicted in Algorithm 2 which is also taken from Ábrahám et al. [33].

Algorithm 2 takes into account the set of samples until the current $i^{\text{th}}$ dimension and all constraints with the main variable $x_i$. The set of constraints is considered to be known. Into each such constraint the given, potentially partial, sample set is substituted (Line 4). It might happen that the constraint is thus fully substituted in which case it can be trivially

handled by evaluating the resulting statement. If the statement is false, the whole domain in this dimension is unsatisfiable for this sample and a covering can be returned (Line 6). As Line 8 shows, fully substituted constraints resulting into a tautology can be skipped as they do not provide unsatisfying intervals. The more intricate case happens if the constraint has $x_i$ as the only remaining variable. In this case the real roots of the polynomial in the constraint are computed. The sign-invariant regions for this polynomial are computed as a point interval for each root and open intervals between them, including unbounded intervals before the first and after the last root as shown in Line 10. For each such sign-invariant interval that violates the constraint, an unsatisfying interval is introduced. The interval includes the information from which polynomial it originated. The details on how the storage depicted in Lines 13 to 16 works will be explained at a later point. In the end, all found unsatisfying intervals are returned.

Back to the main Algorithm 1, Line 2, the computed set of unsatisfying intervals is checked for coverings. If such a covering exists, the algorithm can return that the current dimension is unsatisfiable with the given set of samples and which intervals are responsible. The algorithm will go back and try to find a different sample for the previous variable in the variable order and add the unsatisfying interval around the previous sample. If a covering has been detected in the first dimension, the set of constraints is unsatisfiable.

Assuming that no covering was found yet, an arbitrary sample outside of the unsatisfying intervals is chosen, i.e. in the intervals of the domain that were not explored yet. If the current variable happens to be the last one in the variable order, the sample set is full and can be returned as a satisfying witness as depicted in Line 5 of Algorithm 1. This condition serves as the termination condition of the recursion as afterwards, the $i$-dimensional set of samples is given to a recursive call of the main algorithm. Hence, the current sample is tested in the higher dimensions for its feasibility. In the satisfying case, the full sample set received from the recursive call can be returned as shown in Line 8.

If no satisfiability has been detected in the higher dimensions, the newly found unsatisfying interval around the current sample has to be determined. For determining such an interval, one first determines a characterization of the unsatisfying region around the sample as depicted in Line 10. The characterization is a set of polynomials that describes this region. This set can be computed using the known unsatisfying intervals because they store which constraints they originated from, see Algorithm 2, lines 13 to 16. Note that if the interval was created out of a characterization in a previous run of the while loop (Algorithm 1, Line 11),

it might originate from more than one polynomial.

To illustrate this process, consider the example formula $(x^2 + y^2 - 1 = 0) \wedge (x^2 \neq 0)$. The variable order in this example is $y < x$, i.e. $y$ is to be assigned a value before a value is assigned to $x$. Assume that in the first dimension, the `get_unsat_intervals` algorithm did not find any unsatisfying intervals for $y$'s dimension and the value $-1$ was assigned to $y$. In the second dimension, `get_unsat_intervals` calculates the unsatisfying point interval $[0, 0]$ from the constraint $(x^2 \neq 0)$. For $y$, -1 is substituted into $(x^2 + y^2 - 1 = 0)$ yielding $(x^2 = 0)$ and the unsatisfying intervals $(-\infty, 0)$ and $(0, \infty)$ are determined. The algorithm returns the covering $\{(-\infty, 0), [0, 0], (0, \infty)\}$. Back in the dimension of $y$, the algorithm tries to find an explanation for the unsatisfiability of the assignment $y = -1$.

The characterization ensures the existence of the lower and upper bounds of the conflict by adding the leading coefficients and discriminants of the originating polynomials of each interval in the covering. The discriminants add the information whether the original polynomials have multiple roots in one point by evaluating the discriminants' zeros. For the example, the algorithm adds the discriminant $y^2 - 1$ from the polynomial $x^2 + y^2 - 1$ for the interval $(-\infty, 0)$. The discriminant of $x^2$ is 0. This discriminant will not be helpful for further computations, for simplicity it will be omitted in the following. The last interval $(0, \infty)$ originated from the same polynomial as $(-\infty, 0)$, so the resulting discriminant is already known. The resulting set of discriminants is $\{y^2 - 1\}$.

The leading coefficients are added to the characterization because they indicate the asymptotes of the original polynomials. Here, the coefficients are preselected according to their usefulness for building the bounds of the unsatisfiable region. In a loop, the leading coefficient of a polynomial is added. If the leading coefficient evaluated at the unsatisfiable sample is zero, the next leading coefficient of the polynomial without the former leading coefficient is determined. If it is non-zero, all following coefficients of the polynomial can be omitted. For the polynomials from which the intervals in the example originated, this will add nothing.

To ensure the bounds of the unsatisfying region are the closest possible ones, intervals include a storage for all polynomials that are define their bounds as sets $L$ and $U$. This includes the singletons stored in Line 14 and Line 15 of Algorithm 2, where the original polynomial is stored if a real bound was found. For intervals that are calculated from a characterization, these defining sets might contain more that one polynomial. Another component stored in the interval is a set $P_i$ of all polynomials with main variable $x_i$ that were used to create the intervals. For Line 16 of Algorithm 2 that is the fifth input of the interval but again the set may contain more than one polynomial. For the example, the polynomial $x^2$ is

stored as a reason for both bounds of the interval $[0, 0]$ and the polynomial $y^2 + x^2 - 1$ is stored as a reason for the higher bound of $(-\infty, 0)$ and the lower bound of $(0, \infty)$.

To use all this information for ensuring the lower bound to be the closest one, one computes a new set of polynomial resultants for each of the unsatisfying intervals as explained in the following.

$$\{ \text{ res}(p, q) \mid p \in L,\ q \in P_i,\ q \text{ has a root smaller than or equal to } l \ \}$$

The set contains all resultants of a defining polynomial of the lower bound, $p \in L$, and one of the polynomials from which the interval originated with the current variable $x_i$ as its main variable. This second polynomial is named $q$ and is only chosen if it has a root below or at the interval's lower bound $l$. This new resultant set is added to the characterization for each interval. In the example only one polynomial is involved in each of the intervals. Hence, these resultants will always be 0 and are omitted. So far, the characterization is still $\{y^2 - 1\}$.

For the upper bound, these sets are calculated analogously as all resultants of a defining polynomial of the upper bound $p \in U$ and an origin polynomial with the main variable $x_i$ that has a root above or at the interval's upper bound $u$.

$$\{ \text{ res}(p, q) \mid p \in U,\ q \in P_i,\ q \text{ has a root larger than or equal to } u \ \}$$

The resultants for the upper bounds are omitted for the example for the same reason as omitting the resultants for the lower bounds.

For each pair of consecutive intervals, the resultants of the polynomials that define their bounds are added, too. For the example, this adds the resultant $y^2 - 1$ from $x^2 + y^2 - 1$ and $x^2$ for the intervals $(-\infty, 0)$ and $[0, 0]$. The resultant is added to the characterization set. The same happens for the intervals $[0, 0]$ and $(0, \infty)$, where the same resultant is calculated. As the characterization is a set, it remains to be $\{y^2 - 1\}$. For more details on why resultants are added, one can refer to [33].

If some other polynomials with main variables on a lower dimension also played a role in the creation of the interval, these were stored in the interval and can be added to the characterization, too. For the example, as $x$ is the main variable for the second dimension in all polynomials, none have to be added.

Finally, the characterization of the unsatisfying region is complete and a new interval can be computed from it as illustrated in Algorithm 1 at Line 11. At this point a new unsatisfying interval is created from the characterization that contains the unsatisfying sample. In consequence, the

algorithm cannot exclude only the exact point of the sample but potentially a larger region. To find the largest known unsatisfying area, all real roots of the polynomials from the characterization are computed. These, as well as positive and negative infinity, are candidates for the interval bounds (or for unboundedness, in case of the infinities). For the example, this results in the candidate set $\{-\infty, -1, 1, \infty\}$ for the characterization $\{y^2 - 1\}$.

The bounds for the new interval are chosen from these candidates as the ones the current dimension's sample lies between or on. The lower interval bound is the largest candidate that is below or equal to the sample and the upper bound is the smallest candidate that is above or equal to the sample. In this context, the negative infinity counts as the smallest candidate and the positive infinity as the largest one. Hence, the interval may be unbounded towards infinity if the sample is not between any two roots. In the example, the interval $[-1, -1]$ is chosen as the newly found unsatisfying interval for $y$.

For future iterations, the interval stores all polynomials from the characterization as responsible for it being unsatisfying. This includes the information which polynomials have roots at the new bounds as the sets $L$ and $U$, which can be used in another iteration of finding a characterization.

To conclude the example, a new value is chosen for $y$ that lies outside of the unsatisfying interval set $\{[-1, -1]\}$. Assume that the algorithm chooses 0. Then this assignment is given to a new call of `get_unsat_cover` on the dimension for $x$. Substituting the new value for $y$ into $(x^2+y^2-1 = 0)$, the algorithm computes the unsatisfying intervals $(-\infty, -1)$, $(-1, 1)$, and $(1, \infty)$. From the constraint $(x^2 \neq 0)$, the interval $[0, 0]$ is added again. As the new unsatisfying intervals do not form a covering, a new value for $x$ can be chosen. Choosing for example $x = -1$, there are valid variable assignments for all variables. The algorithm can deduce the satisfiability of the input formula with the satisfying assignment $\{y = 0, x = -1\}$.

# 5 Implementation

Implementing the Cylindrical Algebraic Covering (CAC) algorithm described in Chapter 4 is realized as a module for the toolbox SMT-RAT. SMT-RAT is written in C++ and the implementation is object-oriented. The code of SMT-RAT is available on GitHub [2].

An SMT-RAT module can be used as a theory solver integrated into a user-defined strategy. For further information on SMT-RAT, one can refer to Chapter 2. The CAC module, following the architecture of SMT-RAT, receives the set of constraints that was handed down from the Satisfiability (SAT) solver in each solving step. The constraint set is a subset of the constraints in the input formula and their negations. The module has to decide whether the conjunction of these constraints is satisfiable. In SMT-RAT, each time the CAC module is called it receives a set of constraints to add and a set of constraints to remove from the former set. For this, each module must implement an `addCore` and a `removeCore` method. These are called for each constraint in the subset.

After these calls, the module's `check` function is called. This function is expected to decide the satisfiability of the given constraint set. The CAC module contains a store for a logical model that can be filled if the given problem is satisfiable. The model contains an assignment of variables to values, which is empty in the beginning. There is also a store for unsatisfiable subsets of constraints that is to be filled if the given problem is unsatisfiable.

Two similar implementations are proposed in this thesis, namely the non-incremental approach of rechecking all constraints on every iteration and the incremental approach. The non-incremental approach was implemented as described in the following section. The incremental approach is designed as an extension of the non-incremental approach but the implementation was not stable yet at the end of the thesis.

## 5.1 Non-Incremental Approach

For the non-incremental approach, the constraints that were received to add and remove are stored by the module such that the currently regarded subset of constraints is known. The module contains a CAC class in which the actual implementation of the CAC algorithm resides. In the call of `check`, the former constraint data in the CAC class is reset and the new set of constraints is inserted. The CAC class stores the constraints and extracts the variables contained in the constraints. As the CAC implementation will assign values to the variables consecutively, it is relevant in which order the variables are. This variable order might have a relevant influence on the complexity of the necessary computations. How this order is determined is outside of the scope for this thesis, hence for this thesis it is assumed that a static variable order is determined by the tool in a black-box fashion.

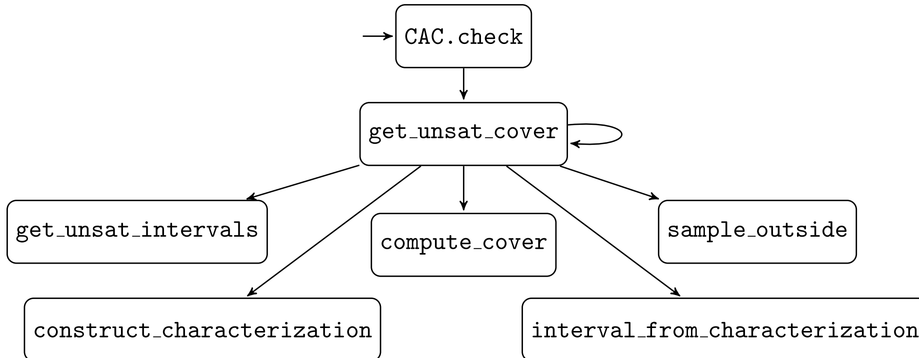
In the following, the implementation of the CAC algorithm that was the main work of this thesis is described. In this chapter, it is often described that the algorithm uses a *sample*, so the term has to be explained. A sample is a set of assignments of values to variables. Given the current dimension, the provided sample of the main algorithm will always contain assignments to all variables of the lower dimensions. The algorithm will try to expand the sample for the current dimension by finding a valid assignment for the current variable.

The implementation works on *intervals* as regions of the domain in the current dimension as described in Chapter 3. As described in the theory of the CAC algorithm in Chapter 4, there is some additional data about the origin of the intervals that has to be stored. Due to this additionally needed information, the existing implementation of intervals in SMT-RAT is not suitable for the CAC implementation. Therefore the CAC implementation requires a new class for intervals. The main items that the intervals have to store are the values for the upper and lower bounds as algebraic numbers. As mentioned in Section 3.1, this is due to the CArL implementation. Additionally, intervals need to store whether the bound is open or closed and, as a special case of open bounds, whether the bound is infinite. To meet the requirements of building a characterization of a conflict as depicted in Algorithm 1 in Line 10 and described in Chapter 4, the intervals need additional store for polynomials. These are stores for polynomials that contribute to the reasons for each bound and stores for contributing polynomials with the dimension's variable as main variable and such ones that have a lower main variable. Additionally, all constraints from which the interval originated are stored, including the ones the mentioned polynomials originated from. The constraint store will be

used for building an explanation set of constraints in case of unsatisfiability, and in the incremental approach for backtracking.

The new intervals have a few handy functions. They allow a trivial check whether they cover the whole domain, as this only happens if both bounds are unbounded towards infinity. The new intervals can also check whether a given value is contained in the interval. For the use in the finding of new samples, the intervals are capable of giving a representative from the interval, i.e. an arbitrary fixed value contained within the interval. For convenience, the new interval class has some additional constructors to input possible subsets of data tailored to the specific applications in the CAC algorithm. As the intervals are to be ordered within the implementation, there is a new comparison operator that determines an interval A to be lower than another interval B if the lower bound of A is lower than the lower bound of B, or in case the lower bounds are equal, if the upper bound of interval A is lower than the upper bound of B. This comparison considers infinities as well as open and closed bounds. This results in an interval order which is consistent with the one in the originating paper of the CAC algorithm [33]. While the CAC algorithm usually works on intervals for which the bounds are either both open or both closed, the new class can technically handle the other cases, too. In the following, the term *interval* will refer to this new interval implementation.

Figure 5.1: Call tree of the CAC algorithm




An overview of the main functions used by the CAC algorithm is shown in Figure 5.1 to guide the reader through the process. The names are similar to the ones used in Chapter 4 and in Algorithm 1 to indicate that they implement the corresponding functions. The implementation is, as suggested in Chapter 4 in the theory, designed in a recursive fashion. This is especially interesting during the backtracking in the incremental approach.

The satisfiability of the entered problem given by the tool is determined by the function `get_unsat_cover`. The current constraints whose conjunction is to be checked are known to the CAC class. The variable order is also known. The function first calls the function `get_unsat_intervals` to get all intervals in which any constraint is unsatisfied. It receives the current set of samples (which is empty in the beginning) and the current variable.

At first the `get_unsat_intervals` function filters the constraints for the ones with the current variable as their main variable as these are the ones relevant in this dimension. This is realized by looping through each constraint's variables and checking whether the main variable is contained and no higher variable in terms of variable order is used.

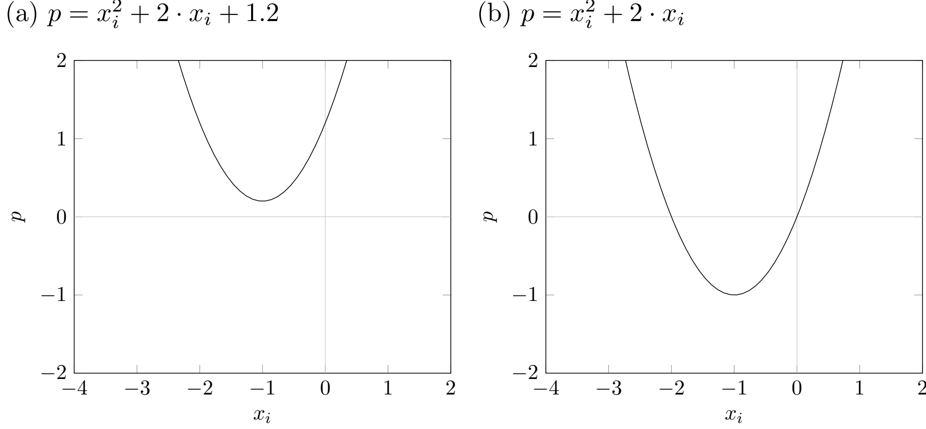
The `get_unsat_intervals` function loops through all constraints that it has found to have the current main variable. Within this loop, it computes all unsatisfying intervals in this dimension by evaluating the constraint with the given sample of dimension $i$-1. This can result in the constraint being false, true, or being ambiguous. The polynomial of the constraint is univariate after the evaluation. This stems from the sample being substituted into the constraint before the evaluation. During this process, all of the variables of lower variable order are replaced by the corresponding assignment in the sample. As the `get_unsat_intervals` function only regards the constraints with main variable $x_i$, there are no variables with higher index, either.

If the constraint evaluates to false, a trivial conflict was found and there is an unsatisfying covering. To illustrate this case, have a look at the polynomial in Figure 5.2a on page 25. The constraint $x_i^2+2\cdot x_i+1.2 < 0$, for example, does present a trivial conflict as the polynomial does not include any values below zero. If this case appears, the set only containing the interval (-$\infty, \infty$) with information about its origin is returned as the new set of unsatisfying intervals.

The second case where the constraint evaluates to true is also trivial as no unsatisfying intervals can be added from the constraint. The example polynomial in Figure 5.2a can be used to illustrate this case, too. Assuming the corresponding constraint is $x_i^2 + 2 \cdot x_i + 1.2 > 0$, the constraint evaluates to true for this dimension.

More intricate is the case in which the constraint does not evaluate to a Boolean value yet. This happens if the polynomial of the constraint with the substituted sample is not sign-invariant. In the theory, this was the case depicted in Algorithm 2 from Line 9 on. An example of such a polynomial is shown in Figure 5.2b. If the shown polynomial $x_i^2 + 2 \cdot x_i$ is the polynomial of a constraint, unsatisfying intervals can be computed. If such a constraint is found, all sign-invariant regions of its polynomial

Figure 5.2: Example polynomials for computing unsatisfiable intervals




are computed by regarding its real roots. These are computed and the regions are determined to be point intervals at these roots and the intervals between each two consecutive roots. Before the first root and after the last one, another interval unbounded towards the corresponding infinity is included in the regions to cover the rest of the domain. For the example polynomial $x_i^2 + 2 \cdot x_i$, the regions are $(-\infty, -2)$, $[-2, -2]$, $(-2, 0)$, $[0, 0]$, and $(0, \infty)$. These regions are designed to be sign-invariant for the constraint with respect to the given sample.

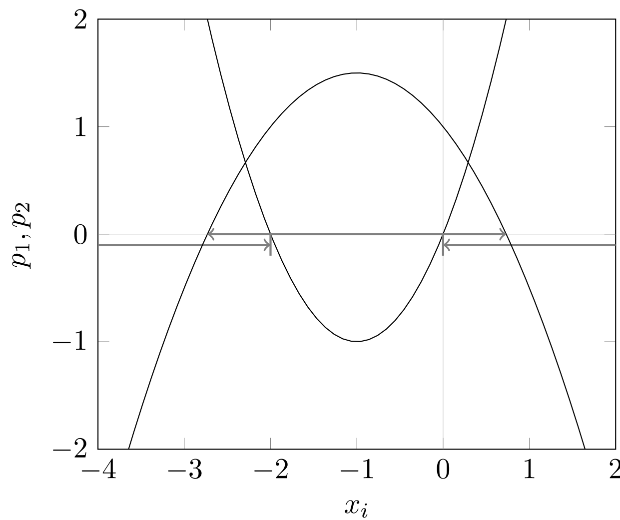
Out of each computed region, the algorithm determines a representative as described in the capabilities of the interval class. This representative, a value within the interval, is assigned to the current dimension's variable. By substituting the constraint with the accordingly extended sample and reevaluating it, one can determine the truth value of the constraint in this region. The regions that are determined to be false by this method are added to the set of unsatisfying intervals. These intervals receive the originating constraint and, for the bounds that are not infinite, the constraint's polynomial as defining the bound. As noted in Chapter 4, this information can later be used to compute further unsatisfying intervals.

When the main algorithm `get_unsat_cover` gets the set of unsatisfying intervals, it enters a loop. This is a direct implementation of the loop described in Chapter 4, see Algorithm 1 in Line 2. The loop condition is, as in the theory, whether a covering of unsatisfying intervals exists. This is done in its own function called `compute_cover`. This function receives the computed intervals.

To illustrate when a covering can be present with regard to the con-

straints, consider the example depicted in Figure 5.3. The figure shows the polynomials of the constraints $x_i^2 + 2 \cdot x_i \leq 0$ and $-0.5 \cdot x_i^2 - x_i + 1 \leq 0$. The constraint $x_i^2 + 2 \cdot x_i \leq 0$ to which the upper polynomial belongs excludes the intervals $(-\infty, -2)$ and $(0, \infty)$. The second constraint, $-0.5 \cdot x_i^2 - x_i + 1 \leq 0$, adds the unsatisfying interval $(-1 - \sqrt{3}, \sqrt{3} - 1)$ which overlaps with the other two intervals, forming a covering.

Figure 5.3: Example for a compound covering using $(p_1 = x_i^2 + 2 \cdot x_i) \leq 0$ and $(p_2 = -0.5 \cdot x_i^2 - x_i + 1) \leq 0$




Coverings can consist of at most the number of intervals in the set of unsatisfying intervals but might not include all of them. The function `compute_cover` can find such coverings. This algorithm was only described informally in the paper by Ábrahám et. al, so the method used in the implementation was developed anew for this thesis. A pseudo code depiction of `compute_cover` is shown in Algorithm 3. To cover the trivial cases, the function first checks whether the interval set is either empty or contains am interval whose two bounds are infinite. If the interval set is indeed empty, it cannot contain a covering (see Line 1). As the intervals have a build-in function to check whether they span the whole domain and therefore form a singleton covering, the check for such is trivial, too. Should such an interval exist, it is returned as a covering (see Line 3).

The remaining part of the `compute_cover` function builds coverings from more than one interval in an iterative fashion. The basic idea is to find an interval unbounded towards negative infinity to start with and expand it with other intervals until the whole domain is covered. It starts by determining whether there is an interval whose lower bound is unbounded

**Algorithm 3:** Pseudo Code: `compute_cover`

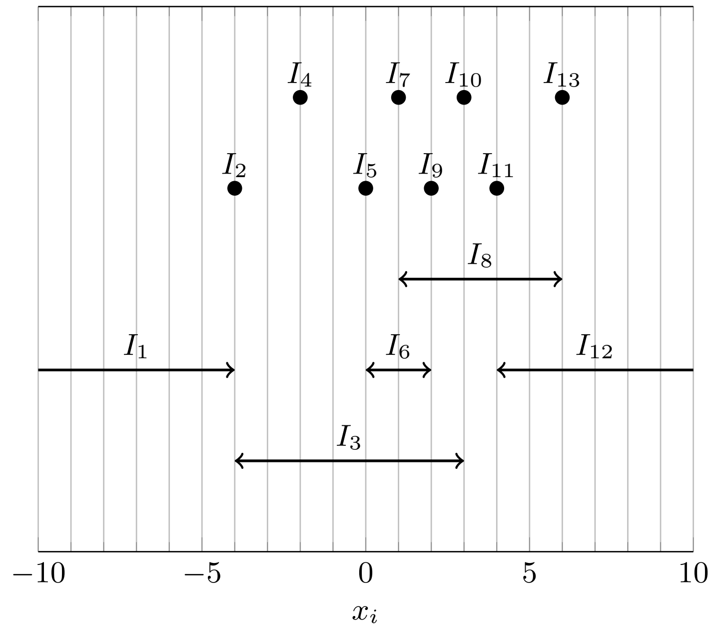
```
   Input  : set of intervals 𝕀
   Output: subset of 𝕀 forming a covering or ∅ if no covering exists
1  if 𝕀 == ∅ then return ∅
2  if 𝕀 contains I with lower bound −∞ and upper bound ∞ then
3  └ return {I}
4  if 𝕀 does not contain an interval with lower bound −∞ then
5  └ return ∅
6  else
7  │ prev := interval with largest upper bound from {I|I ∈ 𝕀 and
   │   lower bound −∞}
8  └ covering.add(prev)
9  for next ∈ I do
10 │ if prev.u == next.u and prev.u open and next.u closed then
11 │ └ covering.add(next); prev := next
12 │ else if prev.u == next.l then
13 │ │ if prev.u open and next.l open then return ∅
14 │ │ covering.add(next)
15 │ │ else if next.u == ∞ then return covering
16 │ └ prev := next
17 │ else if next contains prev.u then
18 │ │ covering.add(next)
19 │ │ if next.u == ∞ then return covering
20 │ └ prev := next
21 └ else return ∅
22 return ∅
```

towards negative infinity. To make this search easier, the intervals are ordered according to the interval 'lower' relation described before. Intervals that feature negative infinity as their lower bound are the lowest one in the order. If there is no such interval, there cannot be a covering as this 'lowest' part of the domain is not covered. At this point the function will return an empty set as a covering to indicate that there is none to be found.

If there are intervals unbounded towards negative infinity, the algorithm chooses the largest one in the interval comparison operator as a starting point. This is the one with the greatest upper bound. As there was a test for singleton coverings beforehand, this upper bound cannot be

unbounded towards positive infinity at this point. To illustrate the process of finding a covering, consider the example shown in Figure 5.4. The points shown in the figure represent closed point intervals. The lines represent intervals with open bounds where the outermost intervals that end without an arrow are unbounded towards the corresponding infinity. As the function `get_unsat_intervals` only returns intervals of such forms, the example was chosen accordingly. Note that the implementation can also handle intervals with mixed open and closed bounds. As the intervals are ordered, the depicted intervals are labelled with their index in the interval order.

Figure 5.4: Example for unsatisfiable intervals that contain a covering




For this example, the algorithm would choose interval $I_1$ as a starting interval as it is the only one unbounded towards negative infinity. This interval is added to the covering, see Algorithm 3 in Line 8. As the intervals are ordered, the function assumes that there is no additional information in intervals that appear before the currently regarded one in the order. As such, it regards the next interval in the order which is interval $I_2$ in the example. Interval $I_1$ was $(-\infty, -4)$, interval $I_2$ is $[-4, -4]$. At first, the function checks whether the upper bound of the last interval is equal to the upper bound of the regarded one as computed in Algorithm 3, Line 10. This is the case for intervals $I_1$ and $I_2$. Due to the order, the lower bound of the regarded interval must be lower or

equal to this value, too. If both bounds of the regarded interval have the same value, as is the case for $I_2$, the interval is a point interval. There is only additional information in such an interval if the upper bound of the last interval is open because the bound was previously excluded. As this is true for $I_1$ and $I_2$, the function can conclude in Line 10 that the interval $(-\infty, -4]$ is covered by the unsatisfying intervals, including the closed bound. Interval $I_2$ cannot give any new information at this point and the function regards the next interval in the order, $I_3$ which is $(-4, 3)$.

Starting anew for this pair of intervals, the intervals $I_2$ and $I_3$ do not have the same upper bounds. The next case the function considers is whether the upper bound of the last interval is equal to the lower bound of the regarded interval as illustrated in Line 12, which fits for $I_2$ and $I_3$. If both of these bounds are open, the exact bound would be excluded. The interval order takes open and closed bounds into account and counts closed lower bounds as 'lower' that open lower bounds. There would not be an interval including the point of the bound in the set if both bounds were open and the function could conclude that the interval set contains no cover in Line 13. As this is not true for $I_2$ and $I_3$, the function can conclude that $I_3$ extends the computed covering. The upper bound of $I_3$ could be unbounded towards infinity, in which case the covering would be complete and could be returned as depicted in Line 15 of the pseudo code. The upper bound is 3, so the function is not finished yet.

It adds $I_3$ to the covering in Line 14 and regards $I_4$ which is $[-2, -2]$. Neither is the upper bound of $I_4$ equal to the one of $I_3$ nor is the upper bound of $I_3$ equal to the lower bound of $I_4$. Therefore, the intervals do not border. The remaining case to consider is whether the upper bound of $I_3$ is contained in $I_4$ in Line 17. Then the intervals would overlap and $I_4$ would according to the interval order add some more of the domain to the covering. This case is not met, so $I_4$ is not included in the covering and the function regards the next interval while keeping $I_3$ as the last interval that added to the covering. The function iteratively finds that neither $I_5$, nor $I_6$ or $I_7$ being $[0, 0]$, $(0, 2)$, and $[1, 1]$ add any new information to the covering.

The next interval to trigger one of the cases is $I_8$ which is $(1, 6)$. The upper bound of the last interval $I_3$ is 3 and contained in $I_8$. In this case the function checks whether the upper bound of $I_8$ is unbounded towards infinity in Line 19, which would complete the covering. As this is not true, the interval $I_8$ is added to the covering and the next interval is checked. Intervals $I_9$, $I_{10}$, and $I_{11}$ are point intervals contained in $I_8$ and as such add no new information. When checking $I_{12}$ which is $(4, \infty)$, the case of the upper bound of $I_8$ being contained in $I_{12}$ is true. This time, the function finds the upper bound of the regarded interval to be unbounded

towards infinity in Line 15. The interval $I_{12}$ completes the covering which can be returned.

Note that the last interval in the order, $I_{13}$ which is $[6, 6]$ is not checked by the function. The covering could be completed beforehand without including it. If there was another interval with its upper bound unbounded towards infinity with a higher lower bound than $I_{12}$, this one would not be added as part of the covering, too.

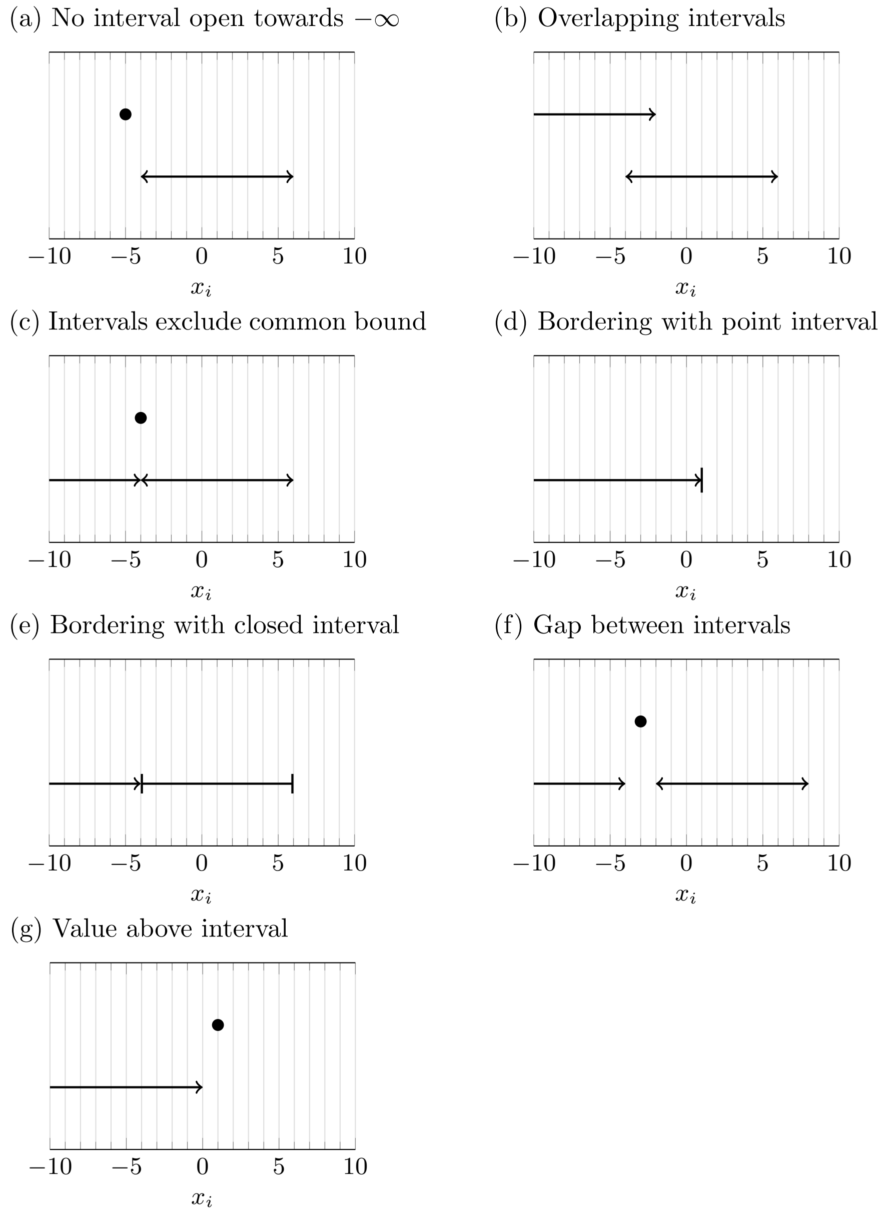
Going back to the main algorithm `get_unsat_cover`, the while loop can with the help of the `compute_cover` function decide whether to continue. As suggested by the theory in Algorithm 1, if a covering was found it is returned. The module extracts all constraints stored as used in the computation of the contained intervals and returns them to the tool as explanation for the unsatisfiability.

If the `compute_cover` function could not find a covering, there are areas of the domain that are not covered by any of the known unsatisfying intervals. A new value from these areas is chosen to be assigned to the current variable. The choosing of a value is done by calling the function `sample_outside` which receives the known unsatisfying intervals as input. Some examples of this process are depicted in Figure 5.5.

The underlying principle is to find the first gap between the unsatisfying intervals and use a value from this gap. The algorithm to find samples is not described formally in the CAC paper [33], so the concept for the `sample_outside` function was newly developed for this thesis. The functioning is similar to the one of `compute_cover`. Therefore the function starts with searching the set of intervals for one unbounded towards negative infinity. If there is no such interval, the function finds the interval with the lowest lower bound and returns an arbitrary value below this bound using functionality from the underlying CArL library. Having a look at the example in Figure 5.5a where the interval with the lowest lower bound is (-4,6). The chosen value in this example is -5. Note that if the set of unsatisfying intervals is empty, this principle will also be applied and the function will return an arbitrary value below 0. If there are intervals that are unbounded towards negative infinity, the function stores the one with the greatest upper bound. Going through the other intervals and using the interval order similarly to the `compute_cover` function, the function distinguishes three cases. The first case applies when the next interval overlaps with the previous one, then the higher upper bound is saved for further comparison as shown in Figure 5.5b.

The second case is the stored bound being equal to the lower bound of the next interval, i.e. the intervals may border to each other. This shared bound can either be excluded from both intervals, which would provide a valid value at this point that can be returned. In the example shown

Figure 5.5: Examples for choosing sample values




in Figure 5.5c, the value -4 is excluded from both intervals and hence a valid assignment for the current variable. The next interval bound could also be a point interval, in which case the bound can be included if it had been previously excluded from the unsatisfying area. For an example of this case one can refer to Figure 5.5d where the point interval [1,1]

adds the point at the value 1 to the previously known unsatisfying area $(-\infty,1)$. The last case of this type of a shared bound applies when the newly regarded interval is not a point interval as shown in the example in Figure 5.5e. This case will not happen in this implementation as the generated intervals have either open bounds or are point intervals. The function handles this case for expandability of the implementation. The interval's information can be added as if the intervals would overlap in this case.

The third case regards the next interval's lower bound being greater than the highest stored unsatisfiable bound. Using the functionality from CArL again, here the function can return an arbitrary value in between these bounds. As an example shown in Figure 5.5f, the value -3 is chosen using the intervals $(-\infty, -4)$ an $(-2, 8)$. If no other cases apply, there is no further interval greater than the currently known highest bound and the function uses the CArL functionalities to return an arbitrary value above this bound. For the example in Figure 5.5g, the chosen value above the bound of the interval $(-\infty, 0)$ is 1.

Having found a new value for the current variable, this assignment is inserted into the sample in the main function `get_unsat_cover`. As extensions to the sample on higher dimensions will be done by a recursive call of the `get_unsat_cover` function, there has to be a termination criterion. For this reason there is a check whether the sample is complete, i.e. whether there is an assignment to all variables. The current variable is the last one in the variable order and there was a satisfying value to assign to it. The function can return this sample as a satisfying witness of the conjunction of the constraints and terminate.

Assuming that the dimension is not the highest one yet, the function tries to complete the sample by doing a recursive call. The new call of `get_unsat_cover` receives the sample including the newly found value and the next variable in the variable order to find a value for. The sample is recursively handed down to the first call of `get_unsat_cover` to return it to the toolbox. If the recursive call returns the sample to be unsatisfiable it found a covering for the next dimension. This covering can be used to find the unsatisfying area around the sample value for the current dimension. To find all relevant polynomials for building this new unsatisfying interval around the current sample point, the function `construct_characterization` is called.

The `construct_characterization` function builds, as the name suggests, a set of polynomials that form a characterization of the unsatisfying area from which a corresponding interval can be built. The implementation of `construct_characterization` stays close to the theory as described in Chapter 4, so this chapter only gives a brief overview of the

implementation-related aspects. To build the characterization, the function uses the polynomials that were stored in the unsatisfying intervals as the origin of the interval and its bounds. As suggested in the theory in Chapter 4, the function goes through all of the intervals in the covering and starts with adding all polynomials with a lower main variable to the characterization. Note that as the intervals are received from the next higher dimension, this might include polynomials with the current variable as their main variable.

To be able to compute the new interval's bounds, the function then works with polynomials with the next higher dimension's variable as their main variable. For the bound computation, discriminants and coefficients of those polynomials are added. As the CArL library includes an operator for computing the discriminants, they can be computed by calling this functionality. For the further computations, the discriminants are normalized and only added if they are not zero. This happens to all added polynomials before being added to the characterization. The coefficients are only added if they are deemed useful for the bound calculation as explained in Chapter 4.

To ensure the bounds that can be computed from the information added so far are the closest possible ones, the function adds resultants as suggested in Chapter 4. For these computations, intervals that are unbounded towards negative or positive infinity are omitted for the computation steps of the corresponding bounds. The CArL library provides the functionalities needed for the computation of resultants. For the computation of the resultants of two consecutive intervals, the interval order is used. When all of these resultants are added, the characterization is complete.

Afterwards, the function `interval_from_characterization` is called on the computed characterization to compute the newly found unsatisfying interval around the current dimensions' current assignment. This, again, is implemented close to the theory in Chapter 4 and only implementation-related aspects are discussed in this chapter. The theory suggests that all polynomials from the characterization are stored within the interval as responsible for the unsatisfiability. In the implementation, the polynomials are split into the set of polynomials that form their irreducible factors before adding them to the characterization. This division into factors is done by the CArL library.

The polynomials with the current variable as main variable are the ones whose roots are potential bounds of the new interval. The function can determine the bounds of the new interval by choosing the two consecutive roots in between which the sample value for the current dimension lies. This is computed by choosing the maximal root smaller than or equal to

the sample value as the lower bound and the minimal root greater than or equal to the sample value as the upper bound. The polynomials that have roots at the respective bound are stored as the polynomials which define the bound. Again, instead of the polynomials themselves, their irreducible factors are stored. It is possible that there are no roots above or no roots below the sample value, in which case the respective bound is unbounded towards infinity instead. If the bounds are equal to the sample, the interval is a (closed) point interval, otherwise the defined bounds are open.

This newly computed interval is added to the set of unsatisfying intervals in the current dimension. As suggested in Chapter 4, the main algorithm `get_unsat_cover` will then restart its loop. The expanded interval set is checked for a covering and if there is none, the function tries to find a new value to assign to the variable. If a covering is found, the function returns this covering instead.

## 5.2 Incremental Approach

The implementation of the incremental approach was not finished during the work of this thesis. However, it can be implemented as described in this section. The incremental implementation extends the non-incremental implementation. It stores intermediate results from within the different dimensions and tries to reuse as much data as possible on the next call of the CAC module. For this reusing, the module holds a store for sets of unsatisfying intervals that are associated with the variable for which they were created. The information about the origin of the intervals that they hold for the algorithm can also be used to react to changes of the constraint set. The model also stores the logical model that was found in its last call, i.e. a satisfying variable assignment, in case the former constraint set was satisfiable. As the SAT solver in SMT-RAT is more advanced than the basic SAT solver described in Section 3.1, it might give smaller subsets of constraints to the theory solver those satisfiability do not necessarily decide the satisfiability of the whole input formula.

The main difference of the incremental implementation is that the CAC module does not have to store the current constraint set as a whole. Instead, constraints that are added by a call of the `addCore` function of the module get checked for consistency. If there is a model from the previous run, the constraint is evaluated for this model. Should the constraint not be satisfied using this model, the stored data has to be updated accordingly. This is done by a crude `backtrack` function that finds out from which dimension on the constraint adds a new restriction on the former

findings. It starts with the first variable in the variable order and evaluates the constraint on this dimensions. That means that the assignment to the first variable in the former model is substituted into the constraint before the evaluation. If the constraint is does not evaluate to false, the assignment for the next variable is added to the substitution and the constraint is reevaluated. This continues until the first variable dimension is found in which the constraint evaluates to false. All stored unsatisfying interval sets from higher dimensions are deleted from the store. These have to be recomputed to adapt to the new conditions. The dimension where the conflict occurred is stored. The `get_unsat_cover` function will start on this dimension on the next call. In case there is more than one backtrack of this kind, the starting dimension for `get_unsat_cover` is set to the lowest detected dimension in which a conflict was detected.

During a call of `removeCore` in the incremental approach, the module does a remove operation on the stored data. This is less complicated than adding constraints as removing constraints has only the potential to remove unsatisfying intervals, not adding new ones. The function removes all unsatisfying intervals that originated from the removed constraint. The former assignment is still satisfiable as this does not impact the satisfiable regions.

In the `check` function of the module, the CAC object is not reset as in the non-incremental approach as they were updated before. The basic structure of the called functions is still the same as in the non-incremental approach. The module's `check` function calls the function `get_unsat_cover`, but gives the lowest conflict dimension as a starting point. If there was no previous assignment, it still starts with the first variable in the variable order. The function uses the known assignment and from the previous run for the lower dimensions. When `get_unsat_inters` is called, it first adds the known intervals to the set of unsatisfying intervals. The function still calculates the unsatisfying intervals from the constraints as these might be newly added. There is an information gain through the intervals from the previous run if intervals were calculated from conflicts before.

Another final difference to the non-incremental approach is in how the `get_unsat_cover` function handles conflicts. Before a characterization is calculated, the stored unsatisfying intervals for the higher dimensions are removed. This insures that further recursion calls do not add unsatisfying intervals that was calculated with the old sample.

# 6 Test results

To get an evaluation of how the Cylindrical Algebraic Covering (CAC) algorithm works in practice, its implementation was tested on instances from the Satisfiability Modulo Theories Library (SMT-LIB) [5]. To have a comparison to define a 'goodness' of the results, the results of the test runs using the new CAC algorithm implementation are compared to SMT-RAT with Cylindrical Algebraic Decomposition (CAD) settings. SMT-RAT is a modular toolbox. It allows the user to specify a custom strategy for the solver. This strategy determines what modules are used and in what order and under which conditions they are applied. For further information on SMT-RAT's strategy, the SMT-RAT manual can be consulted [2]. For the CAC implementation, the only other used module is the Satisfiability (SAT) module which works as a SAT solver as described in Chapter 3. The theory solver consists only of the CAC module. This CAC strategy was tested in comparison of a run of the toolbox using only the SAT module and SMT-RAT's CAD module implementation using SMT-RAT's Lazard projection operator. For the test of the non-incremental CAC approach, the compared CAD strategy is also non-incremental.

As of the end of this thesis the incremental implementation is not stable yet, the resulting intermediate results are not informative yet. Therefore there is no test of the incremental CAC implementation in this thesis.

The implementation of the CAC algorithm was tested on the QF_NRA instance set from the SMT-LIB, the set containing instances over quantifier-free non-linear real arithmetic (QFNRA), from 2019 [5]. This set includes 11,489 instances. The time limit of the solver was set to 60 s with a tolerance of about 3 s. The data stems from runs on AMD Opteron 6172 processors, with a total of 4GB RAM per benchmark. The benchmark was run on each strategy and the computation time and results were measured. A benchmark consisted of a run of the solver on each instance of the QF_NRA library.

The results of using the non-incremental implementations are presented first. An overview of the results using the non-incremental approach is depicted in Table 6.1. The run on the CAD strategy of SMT-RAT ran

| | CAD Strategy | | | CAC Strategy | | |
|---|---|---|---|---|---|---|
| Result | # | % | av. time | # | % | av. time |
| unsat | 3,348 | 29.14 | 0.991 | 4,032 | 35.09 | 1.783 |
| sat | 4,264 | 37.11 | 0.238 | 4,290 | 37.34 | 0.304 |
| total solved | 7,612 | 66.25 | 0.569 | 8,322 | 72.43 | 1.020 |
| timeout | 3,772 | 32.83 | 63.052 | 3,064 | 26.67 | 63.058 |
| memout/error | 105 | 0.91 | 32.546 | 103 | 0.90 | 31.333 |

Table 6.1: Statistics of non-incremental runs on SMT-LIB QF_NRA library with 11,489 instances, times in s, timeout 60 s.

into this limit on 3,772 of the instances which are about 32.83% of the instances. For the CAC strategy, only 3,064 instances ran into the time limit which is about 26.67% of the instances. Hence, the CAC implementation was able to solve more than 6% of instances more than the CAD approach within the time limit.

Other instances that could not be solved usually ran into memory problems, there was only one instance that ran into an memory error in the CAD strategy run, so this is included in the memory section. Using the CAD strategy, the solver ran into memory problems in only about 0.91% of cases and 0.90% of instances in the CAC implementation. Of the 105 impacted runs, 101 instances ran into these problems for both strategies. These 101 instances are omitted for the rest of the evaluation.

Of the 11,489 instances, the solver with the CAD strategy could detect 4,264 to be satisfiable. The CAC strategy could compute 4,290 instances to be satisfiable, which is a slight increase in comparison to the CAD strategy. The average solving time for these solved cases increased a bit, from 0.238 s to 0.304 s.

The original inspiration for the CAC included that it might result in a speed-up in comparison to the CAD algorithm on unsatisfiable problems. The CAD algorithm is part of the CAD strategy. Using SMT-RAT's CAD strategy, the solver could determine the unsatisfiability of 3,348 instances. In contrast to that, the CAC implementation could detect 4,032 unsatisfiable instances within the given time limit. That means that the CAC implementation could detect unsatisfiability in nearly 6% more of the instances than the CAD strategy. On average, the CAC module took 1.783 s for these instances while the CAD solver took only 0.991 s. This is an average increase of 0.792 s.

Looking at the data above, one can already gather that more instances could be solved using the CAC strategy in comparison to the CAD within

the time limit. That does not only include the suspected unsatisfiable instances but a few satisfiable instances, too. As the number of instances that run into the time limit have decreased in comparison to the CAD, the CAC strategy could even solve some instances more than the CAD. The total number of instances solved with the CAC module is 8,322 while the CAD solver could solve 7,612. Hence in total, the CAC implementation solved about 6% more instances than the CAD.

More detailed information about the benchmark of the non-incremental

| Comparison on all 11,489 instances | | | |
|---|---|---|---|
| Result | # | % | av. time difference |
| unsat | +684 | +5.95 | +0.792 |
| sat | +26 | +0.23 | +0.066 |
| total solved | +710 | +6.18 | +0.451 |
| timeout | -708 | -6.16 | -0.006 |
| memout/error | -2 | -0.02 | -1.214 |
| **Instances with the same result for both strategies** | | | |
| Result | # | % | av. time difference |
| unsat | 3,269 | 28.45 | -0.388 |
| sat | 4,112 | 35.79 | +0.046 |
| total solved | 7,381 | 64.24 | -0.146 |
| timeout | 2,830 | 24.63 | +0.010 |
| **Instances only solvable or timeout on CAC strategy** | | | |
| Result | # | % | av. time |
| unsat | 763 | 6.64 | 7.296 |
| sat | 178 | 1.55 | 3.071 |
| total solved | 941 | 8.19 | 6.497 |
| timeout | 234 | 2.04 | 63.006 |
| **Instances only solvable or timeout on CAD strategy** | | | |
| Result | # | % | av. time |
| unsat | 79 | 0.69 | 5.406 |
| sat | 152 | 1.32 | 2.954 |
| total solved | 231 | 2.01 | 3.792 |
| timeout | 942 | 8.20 | 63.051 |

Table 6.2: Statistical evaluation of CAC strategy in comparison to CAD strategy on SMT-LIB QF_NRA library, non-incremental runs, timeout 60 s, times in s.

implementation are depicted in Table 6.2. In the first part of the table, the differences of the two solver strategies are shown again. They are shown as positive if the number increased for the CAC implementation in comparison to the CAD and negative if the number decreased, respectively. The decrease in instances running into the time limit is about equal to the increase in solved instances. To get more information, the rest of the table is divided into instances that were solved for both strategies and such that were only solved in one of them. The percentages in the table were calculated referring to the full set of 11,489 instances as 100%.

The instances that run into the time limit in both cases account for most of the timeout runs happening for the CAC strategy. About 2% of instances ran into the limit in the CAC implementation while being solved by the CAD solver. In contrast to the 8% of instances that could not be solved by the CAD strategy, this number is relatively low.

On the instances that could only be solved by the CAC strategy within the time limit, the CAC implementation took more time for the solving process that the CAD solver needed for the instances that it solely solved. Remarkably, the instances solved as unsatisfiable by both strategies took less time to solve on average using the CAC strategy. The data suggests that there is, in fact, a speed-up for the instances that the solver was able to solve in this time limit with its CAD strategy. Hence, the original aim of the algorithm is met for instances with relatively low solving time.

The average solving time for satisfiable instances solved by both SMT-RAT variants increased minimally by 0.046 s. This increase is relatively low. The percentage of satisfiable instances that could be solved by only one of the strategies is similar for both instances, so there is no relevant advantage or disadvantage to using the CAC implementation on satisfiable instances.

In general, the average solving times of instances that could solely solved by one of the strategies is higher than the average times shown in Table 6.1. A reason for that phenomenon could be that the instances solved by only one of the strategies contained special cases the respective strategy was better suited for. The non-incremental CAC strategy could solve more than 8% of the instances that the CAD strategy was not able to solve in the given time. It can be concluded that the strategy is better suited to solve these instances in QFNRA than the corresponding CAD SMT-RAT strategy.

As mentioned in Chapter 5, there was no stable implementation of the incremental CAC approach at the end of the thesis. Accordingly, there are no test results for this approach.

# 7 Conclusion

In this thesis, the Cylindrical Algebraic Covering (CAC) algorithm described in the paper by Ábrahám, Davenport, England, and Kremer [33] was implemented. There are two approaches to implement the CAC algorithm, a non-incremental and an incremental approach. Both implementations were designed as a theory module for the Satisfiability (SAT) Modulo Theories (SMT) solving toolbox SMT-RAT. The incremental implementation was not finished but could be build as an extension of the non-incremental implementation. To get a notion of how good the non-incremental implementation works, both implementations were tested by running benchmarks on them using suitable instances from the Satisfiability Modulo Theories Library (SMT-LIB). The results were compared on runs of the same instances against SMT-RAT's Cylindrical Algebraic Decomposition (CAD) implementation with Lazard projection.

In conclusion for the non-incremental approach, the experiments indicate that using the CAC implementation instead of the non-incremental CAD strategy might actually result in a speed-up on unsatisfiable instances. For instances with short solving time, a small speed-up could be observed. As the CAC strategy could solve a relevant percentage of instances that the CAD strategy could not solve within the time limit, it can be concluded that these instances would take more time on the CAD settings. Hence, while not directly visible in the presented data set, there is in fact a speed-up on the tested instance set. As the solving times and rates for satisfiable instances for the CAC strategy were similar to the ones to the CAD strategy, there is no disadvantage in using the CAC strategy for such instances. In result, the incremental CAC strategy is time-wise better suited to solve the given instances in quantifier-free non-linear real arithmetic (QFNRA) than the CAD strategy.

As an outlook, the future work will include stabilizing the incremental CAC implementation. There are also some more optimizations that can be applied to the implementation of the CAC algorithm. These include, for example, adding more data to extend the incremental approach. If the implementation would keep more information about which constraints were

newly added, the `get_unsat_intervals` function could omit computing unsatisfying intervals for all formerly known constraints. This might need more information about whether the intervals of this level were computed before. Currently, this is not stored. An empty set of unsatisfying intervals may mean that they were deleted, or not computed before, or that the computation did find no unsatisfying intervals for this dimension. Storing and using these information could save some computation time.

Another possible optimization to the implementation in general is the merging of the basic functionalities used in the functions `compute_cover` and `sample_outside` that were described in Chapter 5. These are similar enough as both search a set of consecutive intervals. Actually, the implementation is easily adaptable to this optimization as the function `compute_cover` was originally designed with the capability to return the last added interval in the case there is no covering. Additionally, the CAC object could store this information and restart both of the algorithms at this last found interval on the next call. Note that for an incremental approach, this would need further adapting when intervals are deleted.

# Glossary

**CAC** Cylindrical Algebraic Covering

**SAT** Satisfiability

**SMT** SAT Modulo Theories

**CAD** Cylindrical Algebraic Decomposition

**NRA** non-linear real arithmetic

**QFNRA** quantifier-free non-linear real arithmetic

**QFNIRA** quantifier-free non-linear mixed integer real arithmetic

**mcSAT** model-constructing satisfiability calculus

**DPLL** Davis–Putnam–Logemann–Loveland

**CDCL** conflict-driven clause learning

**NuCAD** Non-uniform Cylindrical Algebraic Decomposition

**SMT-LIB** Satisfiability Modulo Theories Library

# Bibliography

[1] CArL git repository. https://github.com/smtrat/carl [Online; accessed 09-February-2020].

[2] SMT-RAT git repository. https://github.com/smtrat/smtrat [Online; accessed 09-February-2020].

[3] Yices git repository. https://github.com/SRI-CSL/yices2 [Online; accessed 09-February-2020].

[4] Z3 git repository. https://github.com/Z3Prover/z3 [Online; accessed 09-February-2020].

[5] SMT-LIB, the satisfiability modulo theories library, 2019. http://smtlib.cs.uiowa.edu/ [Online; accessed 04-February-2020].

[6] The Yices SMT solver, 2020. https://yices.csl.sri.com/ [Online; accessed 02-February-2020].

[7] SMT Workshop 2018 affiliated with Federated Logic Conference 2018. 13th international satisfiability modulo theories competition (SMT-COMP 2018), results QF_NIRA, 2018. http://smtcomp.sourceforge.net/2018/results-QF_NIRA.shtml [Online; accessed 27-January-2020].

[8] SMT Workshop 2019 affiliated with the 22nd International Conference on Theory and Applications of Satisfiability Testing. 14th international satisfiability modulo theories competition (SMT-COMP 2019), results QF_LIA, 2019. https://smt-comp.github.io/2019/results/qf-lia-incremental [Online; accessed 02-February-2020].

[9] SMT Workshop 2019 affiliated with the 22nd International Conference on Theory and Applications of Satisfiability Testing. 14th international satisfiability modulo theories competition (SMT-COMP 2019), results QF_NIRA, 2019. https://smt-

comp.github.io/2019/results/qf-nira-single-query [Online; accessed 12-February-2020].

[10] SMT Workshop 2020 affiliated with the International Joint Conference on Automated Reasoning 2020. 15th international satisfiability modulo theories competition (SMT-COMP 2020), 2020. https://smt-comp.github.io/2020/ [Online; accessed 30-January-2020].

[11] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical algebraic decomposition I: the basic algorithm. *SIAM J. Comput.*, 13(4):865–877, 1984.

[12] Christopher W. Brown. Simple CAD construction and its applications. *J. Symb. Comput.*, 31(5):521–547, 2001.

[13] Christopher W. Brown. Constructing a single open cell in a cylindrical algebraic decomposition. In Manuel Kauers, editor, *International Symposium on Symbolic and Algebraic Computation, ISSAC'13, Boston, MA, USA, June 26-29, 2013*, pages 133–140. ACM, 2013.

[14] Christopher W. Brown. Open non-uniform cylindrical algebraic decompositions. In Kazuhiro Yokoyama, Steve Linton, and Daniel Robertz, editors, *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2015, Bath, United Kingdom, July 06 - 09, 2015*, pages 85–92. ACM, 2015.

[15] Christopher W. Brown and Marek Košta. Constructing a single cell in cylindrical algebraic decomposition. *J. Symb. Comput.*, 70:14–48, 2015.

[16] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H. Barkhage, editor, *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer, 1975.

[17] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 360–368. Springer, 2015.

[18] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[19] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.

[20] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[21] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. Sat-based image computation with application in reachability analysis. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2000.

[22] Hoon Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In Shunro Watanabe and Morio Nagata, editors, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '90, Tokyo, Japan, August 20-24, 1990*, pages 261–264. ACM, 1990.

[23] Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals OR*, 181(1):89–107, 2010.

[24] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Confer-*

*ence, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[25] Daniel Lazard. An improved projection for cylindrical algebraic decomposition. In Chandrajit L.Bajaj, editor, *Algebraic Geometry and its Applications: Collections of Papers from Shreeram S. Abhyankar's 60th Birthday Conference*, pages 467–476. Springer New York, 1994.

[26] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition. In B. F. Caviness, editor, *EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 2: Research Contributions*, volume 204 of *Lecture Notes in Computer Science*, pages 277–278. Springer, 1985.

[27] Scott McCallum, Adam Parusinski, and Laurentiu Paunescu. Validity proof of Lazard's method for CAD construction. *J. Symb. Comput.*, 92:52–69, 2019.

[28] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

[29] Conference on Automated Deduction. Herbrand award for distinguished contributions to automated reasoning, 2019. http://www.cadeinc.org/Herbrand-Award [Online; accessed 27-January-2020].

[30] European Joint Conferences on Theory & Practice of Software. ETAPS 2018 test of time award, April 2018. https://etaps.org/2018/test-of-time-award [Online; accessed 27-January-2020].

[31] Alfred Tarski. A decision method for elementary algebra and geometry. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84. Springer Vienna, 1998. Reprint with permission from University of California Press, Berkeley and Los Angeles, 1951.

[32] Boyan Yordanov, Youssef Hamadi, Hillel Kugler, and Christoph M. Wintersteiger. Z3-4biology SMT-based analysis of biological computation. Technical Report MSR-TR-2012-31, March

2012. https://www.microsoft.com/en-us/research/publication/z3-4biology-smt-based-analysis-of-biological-computation/ [Online, Accessed 09-February-2020].

[33] Erika Ábrahám, James H. Davenport, Matthew England, and Gereon Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings (preprint). *Journal of Logical and Algebraic Methods in Programming*, 2020. https://publications.rwth-aachen.de/record/782464 [Online, Accessed 09-February-2020].