The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

EFFICIENT DATA STRUCTURES FOR CYLINDRICAL ALGEBRAIC COVERINGS

Philip Kroll

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

Additional Advisor:
Jasper Nalbach

Aachen, Date

October 26, 2020

Abstract

The *Cylindrical Algebraic Covering (CAC) algorithm* is a decision procedure that is used to decide the satisfiability of problems from the theory for *non-linear real arithmetic*. It is intended for use in *Satisfiability Modulo Theories (SMT)* solving, which is often used in Verification of Hardware and Software. However, this approach underperforms when compared to other decision procedures in the same setting. In this thesis, two changes to the CAC algorithm are presented and implemented with increased performance in mind. These changes include a data structure to store and reuse calculations on polynomials and the addition of incrementality.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Contents

1 Introduction **9**

2 Preliminaries **11**
2.1 Boolean Satisfiability Problem 11
2.2 Real Arithmetic 12
2.3 Satisfiability Modulo Theories 14
2.4 Cylindrical Algebraic Covering 14

3 Algorithm **21**
3.1 Projection Memory 21
3.2 Incrementality 22
3.3 Example 27

4 Implementation **35**
4.1 Projection Memory 36
4.2 Incrementality 37

5 Test Results **41**
5.1 Projection Memory 41
5.2 Incrementality 45
5.3 Incrementality and Projection Memory 46

6 Conclusion **49**

Bibliography **51**

Chapter 1

Introduction

Computer systems are used in areas where correctness and certainty that a particular property holds are required. Correctness and certainty can be acquired by modeling the system in an abstract way to an arithmetic formula and formally reason that those particular properties of the system hold. These properties can often be reduced to questions such as *"Is there a variable assignment, such that the formula evaluates to true?"*. Depending on the problem, a specific underlying theory is used to model the formula. SMT-Solvers are then used to answer these kinds of questions. The *Cylindrical Algebraic Covering (CAC)* algorithm, recently introduced by Ábrahám, Davenport, England, and Kremer [ÁDEK20] can be used in SMT-Solvers for solving quantifier-free non-linear real arithmetic (QF_NRA). This algorithm was implemented over the course of a master thesis in SMT-RAT [Fra20]. Over the course of this thesis, we will present two modifications to the CAC algorithm with an increase of performance in mind. For this purpose, a data structure is introduced, which stores calculations on polynomials so that they can be reloaded if necessary without recalculating them. Additionally, the algorithm is adapted to work incrementally. The intermediate information of the algorithm is stored so that this information can be reused in the next call of the algorithm. The motivation of these modifications is to reduce the number of repeated, time-consuming calculations. The modifications can be used individually, but also in combination. These modifications are also applied to the existing implementation of the CAC algorithm.

In Chapter 2, the theoretical basis for this work is presented. Satisfiability Modulo Theories (SMT) with the theory of real arithmetic and the working of the CAC algorithm are presented. Subsequently, in Chapter 3, the proposed modifications are discussed in more detail. In Chapter 4 of this thesis, the implementation of the modification in the SMT Solver SMT-RAT [CKJ+15] is presented. In Chapter 5, the results of the changes are evaluated and discussed. Finally, the work is briefly summarized in Chapter 6, followed by a summary of the results and an outlook for future improvements.

Chapter 2

Preliminaries

In this chapter, the necessary theoretical knowledge for this thesis is explained. First, the *Boolean Satisfiability Problem* (SAT) is introduced. Afterwards, the theory of interest for this thesis, *Non-linear Real Arithmetic* (NRA) is described followed by an introduction of *Satisfiability Modulo Theories* (SMT). For a better understanding of the main part, the *Cylindrical Algebraic Decomposition* (CAD) and the *cylindrical algebraic covering* (CAC) are introduced which are decision procedures for deciding NRA.

2.1 Boolean Satisfiability Problem

The *Boolean satisfiability problem* (SAT) is the problem of determining if a given formula in propositional logic is satisfiable. A *propositional logic formula* consists of a fixed set of *atomic propositions* to which the values of the boolean *constants* {0,1} can be assigned. A propositional logic formula can then be constructed with atomic propositions and the *logical connectives* for negation $\neg$ and conjunction $\wedge$.

Definition 2.1.1. *Propositional Logic Formula*

$$\varphi := \quad x \quad | \quad \neg\varphi \quad | \quad (\varphi \wedge \varphi)$$

where x is a atomic proposition.

Additional logical connectives such as $\{\vee, \rightarrow, \leftrightarrow, \oplus\}$ can be defined based on the given logical connectives as syntactic sugar. In the following let Φ denote the set of all propositional logic formulae.

Example 2.1.1. *Propositional Logic Formulae*

$$\varphi_1 := (x \wedge y)$$
$$\varphi_2 := (x \vee y) \rightarrow z$$
$$\varphi_3 := \neg z$$

With atomic propositions x,y,z and $\varphi_1, \varphi_2, \varphi_3 \in \Phi$.

A *variable assignment* then assigns boolean constants to the respective variables and the formula can be evaluated. A propositional logic formula is *satisfiable* if a variable assignment exists such that the formula evaluates to true. If such an assignment does not exist the formula is *unsatisfiable*. The boolean satisfiability problem is

NP-complete [Coo71]. Algorithms which solve the boolean satisfiability problem are known as *SAT-Solvers*.

2.2 Real Arithmetic

In this section, the theory of *real algebra*, often also called *non-linear real arithmetic* (NRA), is introduced. It is a first order logic theory over $\mathbb{R}$, also called the *reals*, together with addition and multiplication. It enables inequalities and equalities over real numbers. The following definitions are taken from [BFT16], [ÁDEK20] and [BPR06].

Definition 2.2.1. *Theory of Real Arithmetic*

Domain:	$\mathbb{R}$
Function symbols:	$\{+, -, \cdot\}$
Comparison predicates:	$\{\geq, >, =, \neq, <, \leq\}$

The theory of real arithmetic is decidable [Tar98]. In this thesis we only consider the quantifier-free fragment *QF_NRA* as defined in [BFT16]. This means that there are no universal quantifiers and no negations of expressions containing existential quantifiers.

Definition 2.2.2. *QF_NRA formulas*
QF_NRA formulas are boolean combinations of polynomial constraints.

$$\begin{aligned} &\textit{Terms:} && t := 0 \mid 1 \mid x \mid t + t \mid t \cdot t \\ &\textit{Constraints:} && t := t < t \\ &\textit{Formulas:} && \varphi := c \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \end{aligned}$$

Additional comparison predicates, such as $\{\geq, >\leq, =, \neq\}$, can be defined based on the given comparison predicate as syntactic sugar. We denote the coordinates of $\mathbb{R}^m$ as x_1,...,x_m for some $m \geq 1$.

Definition 2.2.3. *Polynomial*
A term of the form

$$P := \sum_{k=0}^{n} a_k \prod_{i=0}^{m} x_i^{e_{k,i}}$$

with exponents $e_{k,i} \in \mathbb{N}_0$, coefficients $a_k \in \mathbb{Q}$ and variables $x_i \in \mathbb{R}$ for $0 \leq k \leq n$ and $0 \leq i \leq m$ is called a polynomial. We assume that the variables are ordered under the total ordering $\prec$ of their labels, i.e. $x_1 \prec x_2 \prec ... \prec x_n$.

Definition 2.2.4. *Main Variable and Level*

- *The main variable of a polynomial is the highest variable in the ordering present on the polynomial.*
- *The level of a polynomial is the position of the main variable in the variable order $x_1 \prec x_2 \prec ... \prec x_n$*

The main variable of a polynomial P is denoted as main(P).

Example 2.2.1. *Main Variable of a Polynomial*

- $P := 4x_1x_2^4 + x_2$: *main*$(P) = x_2$
- $Q := 9x_1x_2 + \frac{17}{5}x_5$: *main*$(P) = x_5$

The set of all polynomials over the real valued variables x_1,...,x_n is denoted as $\mathbb{R}[x_1,...,x_n]$. A polynomial is *univariate* if it contains one variable, i.e it is in $\mathbb{R}[x_i]$ for some variable x_i. A polynomial is *multivariate* if it contains more than one variable, i.e it is in $\mathbb{R}[x_1,...,x_n]$ for $x > 1$. A multivariate polynomial in $\mathbb{R}[x_1,...,x_n]$ can also be interpreted as an univariate polynomials in $\mathbb{R}[x_1,...,x_{n-1}][x_n]$, where x_n in the main variable of the polynomial.

Definition 2.2.5. *Degree*
The degree of a polynomial p is

$$deg(p) = \max_{0<k<n} \sum_{i=0}^{m} e_{k,i}$$

Polynomials can then be used to define *polynomial constraints*. This is an equality or an inequality of a polynomial compared to zero.

Definition 2.2.6. *Polynomial constraint*
An expression

$$p \bowtie 0$$

with $p \in \mathbb{R}[x_1,...,x_n]$ for some $n \geq 1$ and $\bowtie \in \{\geq, >, =, \neq, <, \leq\}$ is called polynomial constraint.

Example 2.2.2. *QF_NRA formula*

$$\varphi := \underbrace{2x_1^2 + 5x_1^3 \cdot x_2 > 0}_{Polynomial\ Constraint} \wedge -x_1 \cdot x_2^4 + 5 \leq 0$$

With the variable ordering $x_1 \prec x_2$. Both polynomials have the main variable x_2

A constraint with a polynomial $c \in \mathbb{R}[x_1, ..., x_n]$ can be evaluated at a (partial) variable assignment. The variables are substituted for the corresponding values. If there are no variables left, the truth value of the (in-)equation is given.

Example 2.2.3. *Evaluation*
Let $c_1 := -x_1 \cdot x_2^4 + 5 \leq 0 \in \mathbb{R}[x_1,x_2]$.

- $c_1(x_1 \mapsto 1) := -1 \cdot x_2^4 + 5 \leq 0 \iff -x_2^4 + 5 \leq 0$
- $c_1(x_1 \mapsto -1, x_2 \mapsto 2) := 1 \cdot 2^4 + 5 \leq 0 \iff 21 \leq 0 \iff$ *False*
- $c_1(x_1 \mapsto 1, x_2 \mapsto 2) := -1 \cdot 2^4 + 5 \leq 0 \iff -11 \leq 0 \iff$ *True*

2.3 Satisfiability Modulo Theories

In this section, *Satisfiability Modulo Theories* (SMT) is introduced. The boolean satisfiability problem is limited by the fact that the used variables may only be binary. However, there are many problems where this is not sufficient. In *Satisfiability modulo theory* (SMT) formulae are boolean combinations of constraints which are defined over some theory. Here the theory of interest is *QF_NRA* which is defined in Section 2.2 Algorithms which solve satisfiability modulo theory problems for given theories are known as *SMT-Solvers*. A SMT-Solver that follows the *DPLL(T) framework* consists of a SAT-Solver and a *Theory Solver*. The SAT-Solver checks the boolean skeleton of the input formula as described in Definition 2.3.1. If a satisfying assignment is found, the set of constraints that are marked as true and the negation of the constraints that are marked as false are passed to the Theory-Solver. The theory solver then checks if the conjunction of the passed constrains is satisfiable in the given underlying theory. In the context of this thesis, the *Cylindrical Algebraic Covering* (CAC) algorithm as described in Section 2.4 is used as a theory solver.

Definition 2.3.1. *Boolean skeleton*
The boolean skeleton $\varphi_{skeleton} \in \Phi$ *of a SMT formula* φ *is obtained by replacing each constraint* p *in* φ *by a fresh propositional variable* X_p.

When a satisfying assignment for the boolean skeleton is found the according set of constraints are passed to the theory-solver. The theory-solver checks if the passed set of constraints is *consistent*, i.e. if a variable assignment exists such that the conjunction of the constraints evaluates to true. Such a satisfying variable assignment is also called a *satisfying witness*. If the constraints are consistent a satisfying witness for the SMT-formula is provided. If the constraints are not consistent a reason for unsatisfiability is provided, which should be the smallest possible set of constraints, which are already unsatisfiable. The SMT-formula is then extended such that similar unsatisfying assignments are ruled out. This is repeated until either a satisfying assignment is found or until enough information has been gathered to conclude unsatisfiability. An illustration of how a SMT-solver works is shown in Figure 2.1. In order for Theory-solvers to work efficiently they should have the following properties:

- **Incrementality**: When SAT has been returned for a given set if constraints and the theory solver is called again for an extended set of constraints, the information that is already known is reused.
- **Infeasible subsets**: Compute a (minimal) subset of the constraints which are already unsatisfiable.
- **Backtracking**: The theory solver should be able to remove constraints in inverse chronological order.

2.4 Cylindrical Algebraic Covering

The *cylindrical algebraic covering* (CAC) algorithm, presented in [ÁDEK20], is a decision procedure for non-linear real algebra and in particular can be used as an Theory-Solver in SMT-solvers for QF_NRA. More precisely, it takes a set of constraints as input and searches for a satisfying variable assignment for the conjunction of these constraints regarding the underlying theory of the reals as described in Section 2.2.

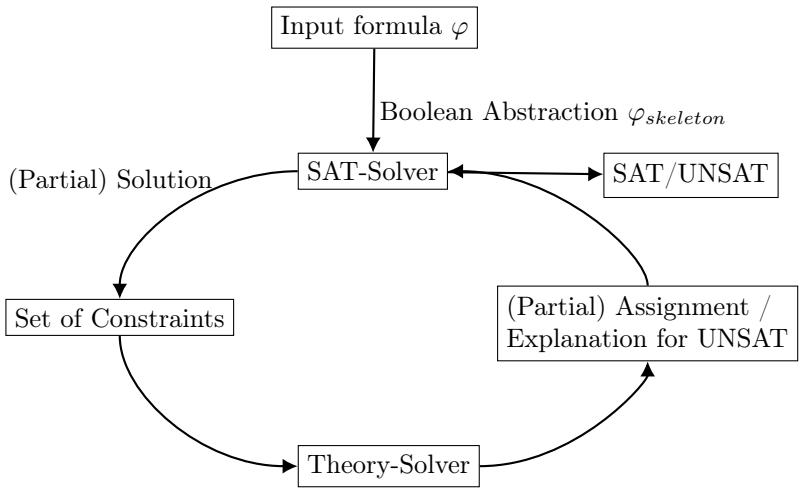

Figure 2.1: Structure of a SMT-Solver

The CAC algorithm is *sound* and *complete*. In case the passed set of constraints is consistent the algorithm returns SAT as well as the corresponding satisfying witness. Otherwise, if UNSAT is concluded, the output includes a subset of the input constraints which conjunction is already unsatisfiable. The CAC algorithm is based on the *cylindrical algebraic decomposition* (CAD) [Col75] and therefore to understand how the CAC algorithm works a fundamental understanding of the CAD algorithm has to be established. A detailed explanation of how the CAD algorithm works can be seen in [Jir95]. The CAD algorithm has a doubly exponential runtime complexity in the number of variables[Col75]. The CAD algorithm determines a finite amount of possible variable assignments, at which the input formula is evaluated. For obtaining the samples, a sign-invariant, cylindrical algebraic decomposition is computed. This decomposition consists of a finite amount of cells, from which one test point each is used as a variable assignment. If none of these assignments is a satisfying witness, it can be concluded that the constraints are not consistent.

Definition 2.4.1. *Cell*

1. *A cell is a non-empty connected subset of* $\mathbb{R}^n$.
2. *A cell is (semi-)algebraic if it can be described as a solution set of a conjunction of polynomial constraints with relation* $\{<, >, =\}$.
3. *A set of cells of* $\mathbb{R}^n$ *is cylindrically arranged if for each pair of cells the projections onto the dimension* $\mathbb{R}^{n-1}$ *are either identical or disjoint.*
4. *A cell is UNSAT for a polynomial constraint if and only if the constraint evaluates to False at every point in the cell.*

The CAD algorithm and likewise the CAC algorithm only considers cells which are both algebraic and cylindrically arranged.

Definition 2.4.2. *Decomposition*
A decomposition D of $\mathbb{R}^n$ *is a finite partitioning into a set of pairwise disjoint cells.*

$$D = \{C_1,...,C_m\} \text{ of } \mathbb{R}^n \text{ with } \mathbb{R}^n = \bigcup_{i=1}^{m} C_i$$

1. *A decomposition is algebraic if each of its cells is algebraic.*
2. *A decomposition is cylindrical if its cells are cylindrically arranged.*

Definition 2.4.3. *Sign-invariance A cell* $C \subseteq \mathbb{R}^n$ *is sign-invariant for a polynomial* $p \in \mathbb{R}[x_1,...,x_n]$ *if exactly one of the following properties holds.*

- $\forall x \in C.p(x) > 0$
- $\forall x \in C.p(x) = 0$
- $\forall x \in C.p(x) < 0$

C is sign-invariant for a set of polynomials if it is sign-invariant for each polynomial.

A decomposition of $\mathbb{R}^n$ is sign-invariant for a set of polynomials if each cell is sign-invariant on the set of polynomials.
The algorithm to construct such a sign-invariant, cylindrical and algebraic decomposition of $\mathbb{R}^n$ is split into three phases.

1. *Projection phase:* The projection phase consists of a number of steps, in each of which a new set of constraints is constructed. The input of the projection phase are multivariate polynomials in $\mathbb{R}[x_1,..., x_n]$. In each step the amount of variables of the polynomials is decreased by one. Polynomials in $\mathbb{R}[x_1, ..., x_{i+1}]$ are mapped to polynomials in $\mathbb{R}[x_1, ..., x_i]$. This is done iteratively until all polynomials are univariate, i.e. are in $\mathbb{R}[x_1]$.

2. *Base phase:* In the base phase, i.e. CAD for $\mathbb{R}[x_1]$, the real roots of the univariate polynomials are isolated. Every root and any point in the interval between roots is chosen as a sample point. These cover all cells of a decomposition of $\mathbb{R}^1$.

3. *Construction phase:* In the construction phase, sample points of $\mathbb{R}^1$ are used to construct the CAD cells of $\mathbb{R}^n$.

In Figure 2.2 an illustration of how the three phases work can be seen. The CAD algorithm first generates all algebraic information of the given set of constraints we want to prove consistency of. The CAD algorithm for constraints in $\mathbb{R}[x_1, ..., x_n]$ first constructs all cells in $\mathbb{R}^n$. From each of these finitely many cells one point, a so called *sample point* is chosen. These sample points are then iteratively used to check the consistency of the constraints. If one of these sample points satisfies all constraints, it is returned and SAT is reported. Otherwise, if for each of the sample points at least one constraint evaluates to False, UNSAT is returned.

The CAC algorithm is based on the CAD algorithm but uses a different approach to generate variable assignments. It is a conflict driven approach where sample points are *guessed* and cells are created incrementally. For a detailed description of the CAC algorithm, please refer to the original paper [ÁDEK20]. The correctness of the CAC algorithm is based on the correctness of the CAD algorithm. The CAC algorithm uses sub-algorithms which are analogous to the phases of the traditional CAD. The

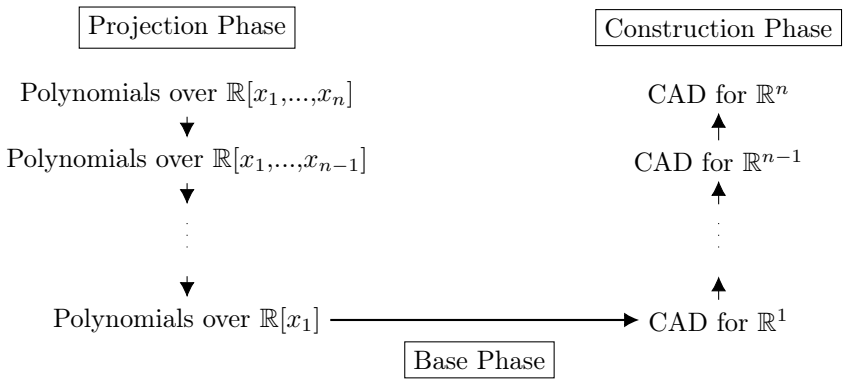

Figure 2.2: Structure of the CAD algorithm

algorithm `construct_characterization`corresponds to the projections phase in the traditional CAD and the algorithm `interval_from_characterization` corresponds to the construction phase in the traditional CAD. The cells are constructed by generalizing the sample point to a cylindrical interval around it. The cells that are created are not necessarily disjoint. Therefore a *covering*, see Definition 2.4.4 is created.

Definition 2.4.4. *Covering*
A covering of $\mathbb{R}^n$ is a finite set of cells $D = \{C_1,...,C_m\}$ of $\mathbb{R}^n$ with $\mathbb{R}^n = \bigcup_{i=1}^{m} C_i$

1. *A covering is algebraic if each of its cells is algebraic.*
2. *A covering is cylindrical if its cells are cylindrically arranged.*
3. *A covering D is UNSAT if every cell is UNSAT for at least one constraint.*

The coverings found by the CAC algorithm are all UNSAT coverings for the given set of constraints. New samples from outside of the existing cells are selected until either a satisfying witness is found or until a set of possibly overlapping cells forms a cover for the entire space. Information about these cells are stored in a data structure I, see Definition 2.4.5.

Definition 2.4.5. *Object I*
Let $s \in \mathbb{R}^{i-1}$ be a (partial) sample point. I represents an interval of $S \times \mathbb{R}$ and carries additional algebraic information. I has six attributes.

- *The lower bound: $l \in \mathbb{R}^i$*
- *The upper bound: $u \in \mathbb{R}^i$*
- *A set of polynomials L with $p(s \times l) = 0$ for all $p \in L$*
- *A set of polynomials U with $p(s \times u) = 0$ for all $p \in U$*
- *A set of polynomials with the main variable x_i: P_i*
- *A set of polynomials with the main variable smaller than x_i: $P_\perp$*

In the following I also denotes the interval $[l,u]$. The sets of polynomials L,U are multivariate polynomials which become univariate when evaluated at S with the bounds l,u as a real root. The sets of polynomials $P_i, P_\perp$ are used for the generalization of S to a region of unsatisfiability. Information about the current covering are stored in a data structure $\mathbb{I}$ which is a set of cells $\{C_1, ..., C_n\}$, each of which are represented by an object of the data structure I. So we have $\mathbb{I} := \{I_1, ..., I_n\}$. When $\cup_{I \in \mathbb{I}_{x_j}} I = \mathbb{R}$ we also say that $\mathbb{I}$ covers $\mathbb{R}$. The main part of the CAC algorithm works by recursively generating UNSAT coverings for the respective dimensions. At the *i-th* recursive depth, the polynomials of level i are being processed. The main loop of the algorithm

Algorithm 1: get_unsat_cover(S) [ÁDEK20]

Data: Constraints over $\mathbb{R}^n$
Input: Sample point $s = (s_1,...,s_{i-1}) \in \mathbb{R}^{i-1}$. If $s = ()$ then $i = 1$
Output: Either (SAT, S) where $S \in \mathbb{R}^n$ is a satisfying witness or $(UNSAT, \mathbb{I})$ where $\mathbb{I}$ is a covering with algebraic information.

```
1  I := get_unsat_intervals(I)
2  while ⋃_{I∈I} I ≠ R do
3  |  s_i := sample_outside(I)
4  |  if i = n then                                  // Found satisfying witness
5  |  |  return (SAT,((s_1,...,s_{i-1}, s_i))
6  |  (f, O) := get_unsat_cover((s_1,...,s_{i-1}, s_i))          // recursive call
7  |  if f = SAT then
8  |  |  return (SAT,O)                                    // Pass on SAT
9  |  else                                          // Generalize s_i to cell
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                                      // Add cell to covering
13 return (UNSAT, I)
```

is in the function `get_unsat_cover`, given in Algorithm 1 which takes a partial sample point $S = (s_1,...,s_{i-1}) \in \mathbb{R}^{i-1}$ for which none of the constraints over $\mathbb{R}^{i-1}$ evaluates to False as input. Either SAT is reported with a full dimensional witness $S \in \mathbb{R}^n$ or $UNSAT$ with $\mathbb{I}$ when S can not be extended to a satisfying sample in $\mathbb{R}^i$. UNSAT is returned for a dimension when a UNSAT Covering is calculated for it. The algorithm works by recursively calling itself with an extended sample point $S = (s_1,...,s_{i-1}, s_i) \in \mathbb{R}^i$ where s_i is a point outside of the intervals defined $\mathbb{I}$. When s is extended to be a satisfying witness it is returned and SAT is returned. Otherwise $\mathbb{I}$ is extended accordingly by the algorithms `construct_characterization` and `interval_from_characterization`. The algorithm `construct_characterization` represents the projection phase of the CAC algorithm. It is called when an UNSAT has been determined for the next higher dimension. Let $\mathbb{I}_{x_{i+1}}$ be an UNSAT Cover for x_{i+1}, this knowledge can be used to exclude not only the current sample point for x_i, but an interval around it. The algorithm `construct_characterization` is used to construct a new set of polynomials which are used to ensure that the UNSAT covering stays valid when the underlying sample point for x_i is generalized. This new set consists partly of resultants and discriminants (see Definition 2.4.6) of polynomials which define the created object I (see Definition 2.4.5). The zero points of the discriminant indicate where the original polynomial has multiple zeros and as such

ensure that the lower and upper bounds continue to exists and no other varieties are spawned. The resultants ensure that the lower and upper bounds are the closest ones possible. These calculations can be complex and the running time depends on the degree of the polynomials used. In the following, these calculations are also referred to as *projections*. The following definitions of the *resultant* and the *discriminant* and the property of the resultant are taken from [BPR06].

Definition 2.4.6. *Resultant and Discriminant*
Let P and Q be two non-zero polynomials:

$$P = a_p X^p + a_{p-1} X^{p-1} + ... + a_0$$
$$Q = b_q X^q + b_{q-1} X^{q-1} + ... + b_0$$

And let P' be the derivative of P. The Sylvester matrix of P and Q is defined as:

$$Syl(P,Q) = \begin{bmatrix} a_p & \dots & & a_0 & 0 & \dots & 0 \\ 0 & a_p & & & a_0 & & \\ \vdots & & \ddots & & & \ddots & \\ 0 & \dots & & a_p & \dots & & a_0 \\ b_q & \dots & & b_0 & 0 & \dots & 0 \\ 0 & b_q & & & b_0 & & \\ \vdots & & \ddots & & & \ddots & \\ 0 & \dots & & b_q & \dots & & b_0 \end{bmatrix}$$

The resultant of P and Q and the discriminant of P is defined as:

$$res(P,Q) = det(Syl(P,Q))$$
$$disc(P) = \frac{(-1)^{\frac{p(p-1)}{2}}}{a_p} res(P,P')$$

Lemma 2.4.1. *Resultant Property*
Let P and Q be two non-zero polynomials. Then it holds that:

$$res(P,Q) = (-1)^{deg(p) \cdot deg(q)} \ res(Q,P)$$

A detailed explanation of how the resultant of two polynomials is calculated and a precise complexity analysis is given in [Duc00].

Chapter 3

Algorithm

In this chapter we present two changes to the CAC algorithms presented in [ÁDEK20]. These changes intend to improve the running time of the CAC algorithm. In Section 3.1 we present how projections can be saved so that they can be used in the future and not have to be recalculated. In Section 3.2 we adapt the CAC algorithm to work incrementally. In order to achieve this, it is shown which information has to be saved or forgotten in which parts of the algorithm. In addition, we show when the stored information based on the added constraints can be deleted or retained.

3.1 Projection Memory

In this section we present a data structure to save projections. To do this, we must look at the projection phase of the CAC algorithm, the algorithm `construct_characterization`. The CAC algorithm (re-)calculates all discriminants and resultants for each variable assignment, although there is a possibility that these calculations have already been done previously. Accordingly, this calculation would be redundant and it would be more efficient to store these calculations and then load them if necessary.

Consider the case that an UNSAT cover for the dimension x_{i+1} is found. This cover gets processed for the dimension x_i and some projections are calculated based on the reasons of unsatisfiability for the dimension x_{i+1}. If a new sample point is now considered for x_i and an UNSAT cover for x_{i+1} is found again, this covering can contain intervals which have the same polynomials as the reason for unsatisfiability as in the previous iteration. Thus, when calculating the characterization for this UNSAT cover partly the same projections as in the previous iteration are calculated. An example when this happens in practice is presented in Section 3.3.

The calculation of the resultant and discriminant is deterministic and independent from the current sample point and can therefore be saved for later use, see Definition 2.4.6. Thus, before a resultant or a discriminant is calculated, one can check whether this calculation has already been carried out and the result is known. In Algorithm 2 we describe such an algorithm to load and store the result of a discriminant calculation accordingly. If the result is stored, it is loaded from the memory and returned. If the result is not stored, it is calculated, stored and returned.
Only the roots of the resultants and discriminants are considered and therefore the sign of these polynomials does not matter because it does not change the roots. It

Algorithm 2: get_disc(p)

Input: Polynomial p.
Output: Polynomial d with $d =$ disc(p).

```
1 if disc(p) is stored, then
2 |  return Load disc(p)
3 else
4 |  d := disc(p)
5 |  Store (p,d)
6 |  return d
```

can thus be used that the resultant of two polynomials remains the same except for possibly the sign if the order of the parameters is reversed. This property is presented in Lemma 2.4.1. So, when the result of a resultant calculation is to be loaded, the order of the parameters should not matter. In Algorithm 3 we describe an algorithm to load and store the resultant when necessary. This works the same way to Algorithm 2 to store the discriminants with the addition that the order for parameters do not matter when loading the according result. Overhead such as memory usage as well as the

Algorithm 3: get_res(p,q)

Input: Polynomials p,q.
Output: Polynomial r with either $r =$ res(p,q) or $r =$ res(q,p).

```
1 if res(p,q) or res(q,p) is stored then
2 |  return Load res(p,q) or res(q,p)
3 else
4 |  r := res(p,q)
5 |  Store (p,q,r)
6 |  return r
```

complexity for insertion, deletion and searching have to be considered. Thus, for practical usage different heuristics can be used. Heuristics for for insertion could be that only calculations which exceed a certain minimum calculation time or from given dimensions are inserted. In addition, to reduce the memory usage, elements that have not been used for a certain time could be deleted.

In Algorithm 4 we formulate the modified `construct_characterization` algorithm using the memory for resultants and discriminants.

In Line 7 the algorithm `required_coefficients` is called. The specifics of this algorithm are out of the scope of this thesis, but the output of this algorithm depends on the current sample point and thus can not be reused in potential future iterations.

3.2 Incrementality

In this section we describe what changes and data structures are necessary in order for the CAC algorithm, as described in [ÁDEK20] to work *incrementally*. The incremental implementation extends the non-incremental implementation. When satisfiability

Algorithm 4: construct_characterization($s, \mathbb{I}$)

Input: Sample point $s = (s_1,...,s_i) \in \mathbb{R}^i$ and data structure $\mathbb{I}$ describing UNSAT covering over s in dimension i+1.

Output: A set of polynomials $R \subseteq \mathbb{R}[x_1,...,x_i]$ that characterizes a region around s that is already unsatisfiable for the same reasons.

```
1  I := compute_cover(I)
2  R := ∅
3  foreach I ∈ I do
4  |  Extract l = I_l, u = I_u, L = I_L, U = I_U, P_{i+1} = I_{i+1}, P_⊥ = I_{P_⊥}
5  |  R := R ∪ P_⊥
6  |  R := R ∪ get_disc(P_{i+1})
7  |  R := R ∪ {required_coefficients(p) | p ∈ P_{i+1})}
8  |  R := R ∪ {get_res(p,q) | p ∈ L, q ∈ P_{i+1}, q(s × α) = 0 for some α ≤ l}
9  |  R := R ∪ {get_res(p,q) | p ∈ U, q ∈ P_{i+1}, q(s × α) = 0 for some α ≥ u}
10 for j ∈ {1, ..., |I| − 1} do
11 |  R := R ∪ {get_res(p,q) | p ∈ U_j, q ∈ L_{j+1}}
12 Perform standard CAD simplifications to R
13 return R
```

of the constraints is concluded, the intermediate results within the different dimensions are stored. When the CAC algorithm is called again, with an extended set of constraints, as much as possible of these intermediate results are reused. Here, we only consider the the case that new constraints are added and no constraints which have been already considered are deleted. To do this, for each dimension the set of determined unsatisfiable intervals must be kept across theory calls. In addition, the satisfying variable assignment found must be saved. In the following we split the set of total constraints according to Definition 3.2.1.

Definition 3.2.1. *Constraints in an incremental setting*
We split the set of constraints into two sets:

- *Processed Constraints P: For the current (partial) sample point, these constraints evaluate to True. The corresponding regions of unsatisfiability are known.*
- *Unprocessed Constraints U: These constraints have not yet been considered.*

A constraint c is said to be processed if $c \in P$. Likewise it is said to be unprocessed if $c \in U$. A constraint must either be processed or unprocessed. Therefore it must always hold that $P \cap U = \emptyset$. When new constraints are added they are unprocessed. If not, one of the unprocessed constraints could contradict the sample point found. Only the unprocessed constraints contain unknown information. Thus, an advantage that the incrementality brings, is that only the unprocessed constraints have to be considered. The unsatisfiable intervals of the processed constraints have been determined in previous calls of the algorithm and can be loaded from the corresponding data structure. Algorithm 5 shows how to store the information for a variable x_j. First the unprocessed constraints with the main variable x_j are added to the set of processed constraints and then removed from the set of unprocessed constraints, i.e. all unprocessed constraints with the main variable x_j become processed. Afterwards

the intervals $\mathbb{I}_{x_j}$ are stored for the variable x_j. Thus, when we store intervals for a variable x_j all constraints with the main variable x_j are processed.

Algorithm 5: store_dimension($x_j, \mathbb{I}_{x_j}$)

Data: Set of Processed Constraints P, Set of Unprocessed Constraints U
Input: Variable x_j, Set of Intervals $\mathbb{I}_{x_j}$
1 $P := P \cup \{c \mid c \in U \text{ with main}(c) = x_j\}$ `// Add to processed`
2 $U := U \cap \{c \mid c \in U \text{ with main}(c) = x_j\}$ `// Remove from unprocessed`
3 Store $\mathbb{I}_{x_j}$ for x_j

Algorithm 6 shows how to remove the stored information for a variable x_j. First the processed constraints with the main variable x_j are added to the set of unprocessed constraints and removed from the set of processed constraints. Afterwards the stored intervals for the variable x_j are removed. Thus, when we remove the intervals for a variable x_j all constraints

Algorithm 6: remove_dimension(x_j)

Data: Set of Processed Constraints P, Set of Unprocessed Constraints U
Input: Variable x_j
1 $U := U \cup \{c \mid c \in P \text{ with main}(c) = x_j\}$ `// Add to unprocessed`
2 $P := P \cap \{c \mid c \in P \text{ with main}(c) = x_j\}$ `// Remove from processed`
3 Delete saved intervals for x_j

In Algorithm 7 we show how the `user_call` algorithm can be implemented to work in an incremental fashion. In the first call of this algorithm or if the previous result was UNSAT, this algorithm behaves exactly like the non-incremental version.

- Line 1 - 3: The previous sample point is rechecked for the unprocessed constraints. If all unprocessed constraints evaluate to true, the previous sample point is returned as a satisfying witness. The unprocessed constraints do not need to be processed and thus no unsatisfying intervals are added. Therefore the unprocessed constraints remain unprocessed.

- Line 4 - 8: We iterate over the unprocessed constraints which evaluated to false for the previous sample point. We delete the of the assignment for the main variable and all saved intervals and assignments for the higher dimensional variables. Since all information about these dimensions is deleted, all constraints that have these as main variables must be marked as unprocessed. This is done by calling Algorithm 6 for all variables of an higher level.

- Line 9: We call Algorithm 8 with the current partial sample point. As the method is recursive we start at the lowest dimension x_0. In this call, the stored intervals for this dimension are considered. We have to start at the lowest dimension, as coverings for higher dimensions could be found and a new variable assignment for the lowest dimension has to be considered.

- Line 10 - 15: Depending on the returned flag of Algorithm 8 we return SAT or UNSAT. If SAT is returned all constraints must be processed and we save the

Algorithm 7: Incremental user_call(S)

Data: Set of Processed Constraints P, Set of Unprocessed Constraints U, Previous Sample Point $S = (s_1, ..., s_n)$, If none exists we set $S = ()$
Output: Either (SAT, S), where $S \in R^n$ is a full dimensional satisfying witness, or (UNSAT, $\overline{C}$), when no such S exists, where $\overline{C} \subset C$ is also unsatisfiable.)

```
1  F := {c | c(S) ≠ True, with c ∈ U}        // Check unprocessed constraints
2  if |F| = 0 then                          // All unprocessed constraints are true
3  |  return (SAT, S)                       // S is still a satisfying witness
4  foreach c ∈ F do
5  |  Let x_i := main(c)
6  |  S := (s_1, ..., s_{i-1})              // Remove assignment for unsat dimensions
7  |  foreach x_j ∈ {x_{i+1}, ..., x_n} do
8  |  |  remove_dimension(x_j)                                  // Algorithm 6
9  (flag, data) := get_unsat_cover(S, x_0)                      // Algorithm 8
10 if flag = SAT then
11 |  Save S := data
12 |  return (flag, data)
13 else
14 |  Save S := ()
15 |  return (flag, infeasible_subset(data))
```

found satisfying witness for future use. If UNSAT is returned all constraints must be unprocessed and we save an empty assignment.

In Algorithm 8 we show how the `get_unsat_cover` algorithm can be implemented to work in an incremental fashion. The dimension which is of interest for the current call is added as an input. We do this to load the correct corresponding saved intervals from memory.

- Line 1 - Line 2: First, we initialize the stored information about the current partial covering. If x_j is not assigned in S we call `get_unsat_intervals` for S. If x_j is assigned in S all unprocessed constraints must evaluate to True. Otherwise the assignment for x_j in S would have been deleted in the `user_call`, see Algorithm 7. Therefore, we call `get_unsat_intervals` for $(s_1, ..., s_{j-1})$, i.e. we remove the assignment for x_j is the passed sample. The algorithm `get_unsat_intervals` is identical to Algorithm 3 from [ÁDEK20] with the exception that only unprocessed constraints with the main variable x_j have to be considered. The processed constraints are not considered, because the corresponding intervals are already stored and then loaded into $\mathbb{I}_{x_j}$ in Line 2. This will produce a set of intervals $\mathbb{I} := \{I_1, ..., I_j\}$ with $I_i \subseteq \mathbb{R}$ such that $S \times I_i$ is conflicting with some constraints for $1 \leq i \leq j$.

- Line 4 - 6: If the current variable of interest, x_j is not assigned in S a new sample point s_j is generated outside of $\mathbb{I}_{x_j}$, i.e. from $\mathbb{R} \setminus (\cup_{I \in \mathbb{I}_{x_j}} I)$. This sample point is then inserted into S such that x_j is now assigned in S. It is necessary to check that x_j is assigned S because a partial sample may be given by the `user_call`. If x_j is already sampled by S it satisfies all processed constraints

Algorithm 8: Incremental get_unsat_cover(S, x_j)

Data: Set of Processed Constraints P, Set of Unprocessed Constraints U
Input: (Partial) Sample point $S = (s_1, ..., s_i) \in \mathbb{R}^i$ such that no constraint evaluated at S is False, Variable x_j which is of interest
Output: Either (SAT, S), where $S \in \mathbb{R}^n$ is a full dimensional satisfying witness, or (UNSAT, $\mathbb{I}_{x_j}$) when S can not be extented to a satisfying sample in $\mathbb{R}^n$ when no such S exists, where $\overline{C} \subset C$ is also unsatisfiable.

```
1  I_{x_j} := get_unsat_intervals((s_1, ..., s_{j-1}),x_j)   // Only consider unprocessed
     constraints
2  Load stored information about dimension x_j into I_{x_j}
3  while ∪_{I∈I_{x_j}} ≠ ℝ do
4      if x_j is not assigned in S then
5          s_j := sample_outside(I_{x_j})
6          S := (s_1, ..., s_i, s_j)
7      if j = n then      // S has full dimension, found satisfying witness
8          store_dimension(x_j, I_{x_j})                          // Algorithm 5
9          return (SAT,S)
10     (f,O) := get_unsat_cover(S, x_{j+1})                  // Recursive call
11     if f = SAT then                                  // O is a satisying sample
12         store_dimension(x_j, I_{x_j})                          // Algorithm 5
13         return (SAT,S)                                         // Pass on SAT
14     else
15         R := construct_characterization(S, O)
16         S := (s_1, ..., s_{j-1})                        // Remove assignment for x_j
17         I := interval_from_characterization(S, s_j, R)
18         I_{x_j} := I_{x_j} ∪ I
19 remove_dimension(x_j)                                          // Algorithm 6
20 S := (s_1, ..., s_{j-1})                                // Remove assignment for x_j
21 return (UNSAT, I)
```

and it must also satisfy the unprocessed constraints, otherwise the assignment for x_j would have been deleted in the `user_call` .

- Line 7 - 9: S satisfies all constraints with main variables x_j which are assigned in S. Thus, if we have $j = n$, S has full dimension, i.e. all variables are assigned in S and a satisfying witness is found. We store the calculated algebraic information about the partial cover $\mathbb{I}_{x_j}$ for the variable x_j a call to Algorithm 5 with the variable if interest x_j and the set of intervals $\mathbb{I}_{x_j}$. This call also sets call constraints with the main variable x_j to processed. Further, SAT together with the satisfying witness S is returned.

- Line 10 : If S does not have full dimension, then there are variables of a higher dimension which are not sampled. The next higher dimension is explored in the recursive call.

- Line 11 - 13: The recursive call returned the SAT flag and a satisfying witness was found. The calculated information about the dimension are stored with a call to Algorithm 5 for x_j and the set of intervals $\mathbb{I}_{x_j}$ and all constraints with the main variable x_j are processed. SAT together with the satisfying witness is passed on.

- Line 13 - 18: The recursive call returned the UNSAT flag the covering for the x_{j+1} dimension. This covering can be used to exclude an interval around s_j. This interval is added to the set of UNSAT intervals $\mathbb{I}_{x_j}$. s_j is removed from the sample point S to resample x_j in the next iteration of the while-loop if $\mathbb{I}_{x_j}$ does not cover $\mathbb{R}$.

- Line 19 - 21: When $\mathbb{I}_{x_j}$ covers $\mathbb{R}$ we exit the while-loop because there is no point left to sample. All calculated information about the current dimension are removed with a call to Algorithm 6 for x_j and all constraints with the main variable x_j are unprocessed. The assignment for x_j has the be removed from S if there is any and UNSAT is returned together with the information about the covering $\mathbb{I}_{x_j}$.

3.3 Example

We provide an example that uses both the projection memory, as described in Section 3.1 and incrementality as described in Section 3.2. All images were created using the software GeoGebra [HBA+20]. We first consider a set of constraints for which SAT is concluded and the some projections are calculated multiple times. Then the set of constraints is extended to demonstrate how SAT can be concluded in the `user_call`. Afterwards the set of constraints is extended further to demonstrate the use of the stored intervals. While the first explanations are detailed and line information is given, these are left out in the course of the example due to repetition. First consider the following set of constraints :

- $c_1 := (x_0 - 1)^2 + (x_1 - 1)^2 + (x_2 - 1)^2 - 1 < 0$

- $c_2 := (x_0 - 1) \cdot (x_1 - 1) \cdot (x_2 - 1) - 1 < 0$

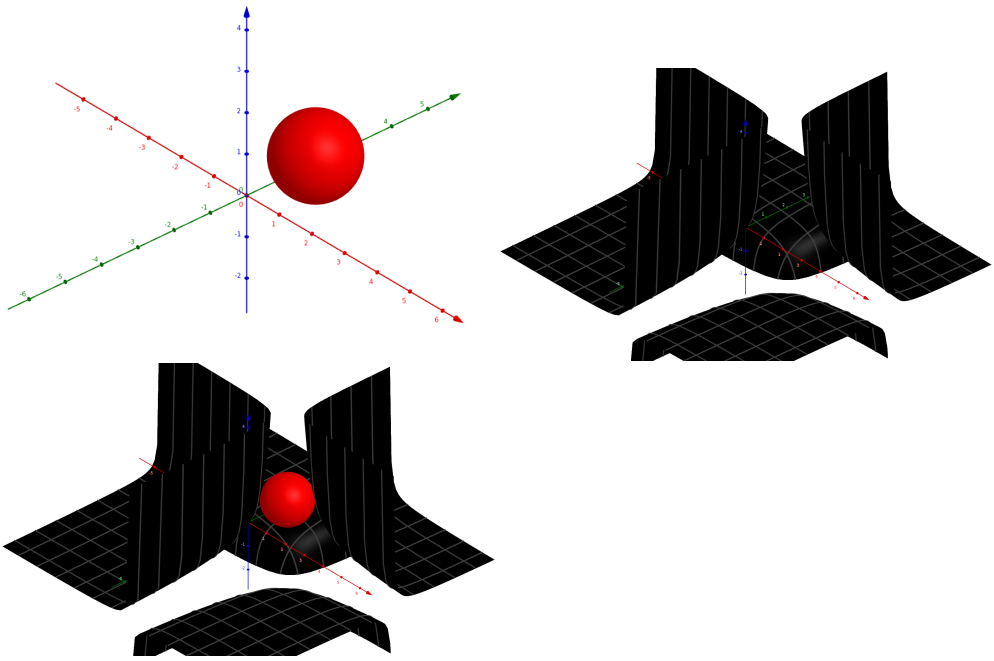

Figure 3.1: Visualisation of the initial constraints. c_1 is presented in red. c_2 is presented in black.

under the variable ordering $x_0 \prec x_1 \prec x_2$. The surfaces defined by the polynomials in the left hand side of the constraints constraints are visualized in Figure 3.1. The red surface represents c_1 and the black surface represents c_2. The constraints are three dimensional and we set $n := 2$. When initially added, the constraints are inserted as unprocessed constraints. So we have $U := \{c_1, c_2\}$ and $P := \emptyset$.

We call the CAC algorithm and jump to the `user_call` as described in Algorithm 7. This is the very first call of the algorithm and we do not have a previously stored sample point S and set $S := ()$. Likewise we do not have any stored information about the variables.

`user_call`$(S = ())$:
The unprocessed constraints $U = \{c_1, c_2\}$ do not evaluate to True at the sample point S since there are still unassigned variables in each case. Thus, we set $F := \{c_1, c_2\}$ in Line 1. With $|F| = 2 > 0$ SAT is not returned in Line 3. We now iterate over the constraints on F. S is empty and there are no saved intervals so there is nothing to remove. Therefore the calls from Line 4-8 have no effect. Afterwards, in Line 9 we call the main algorithm as given in Algorithm 8 for the lowest given variable in the ordering.

`get_unsat_cover`$(S = (), x_0)$:
The current variable of interest x_0 is not assigned and `get_unsat_intervals` is called in Line 1. No intervals of unsatisfiability, as defined in Definition 2.4.5, can be concluded as none of the unprocessed constraints have the main variable x_0 and therefore we set $\mathbb{I}_{x_0} := \emptyset$. There is no stored information for x_0 so there

is nothing to load and add in Line 2. $\mathbb{I}_{x_0}$ does obviously not cover $\mathbb{R}$ and we enter to while-loop. x_0 is not assigned in S so we choose a sample for x_0 outside of $\mathbb{I}_{x_0}$ in Line 5. We set $s_0 := x_0 \mapsto -1$ and add the assignment to S. We have $j = 0 \neq 2 = n$ so S does not have full dimension and we skip the if statement from Line 7 - 9 and jump to the recursive call in Line 10 to explore the next higher dimension.

`get_unsat_cover`$(S = (x_0 \mapsto -1), x_1)$:
x_1 is not assigned and `get_unsat_intervals` is called where no intervals can be concluded as none of the unprocessed constraints have the main variable x_1 and we set $\mathbb{I}_{x_1}$. There is no stored information for x_1 so there is nothing to load and add to $\mathbb{I}_{x_1}$. As $\mathbb{I}_{x_1}$ does not cover $\mathbb{R}$ we enter the while-loop. x_1 is not assigned in S so we sample $s_1 := x_0 \mapsto -1$ and set $S := (x_0 \mapsto -1, x_1 \mapsto -1)$. S does not have full dimension and we jump to the recursive call to explore the next higher dimension.

`get_unsat_cover`$(S = (x_0 \mapsto -1, x_1 \mapsto -1), x_2)$:
Again, the current variable of interest is not assigned and `get_unsat_intervals` is called in Line 1. All unprocessed constraints have the main variable x_2 and have to be considered. $\mathbb{I}_{x_2} := \{I_1\}$ is returned with :

$$I_1 : l := -\infty, u := \infty, L := \emptyset, U := \emptyset, P_2 := \{c_1\}, P_\perp := \emptyset \tag{3.1}$$

There is no stored information for x_2 so there is nothing to load and add to $\mathbb{I}_{x_2}$. $\mathbb{I}_{x_2}$ covers $\mathbb{R}$ and we therefore skip the while-loop and jump to Line 19 and call Algorithm 6. In `remove_dimension` for the variable x_2 the processed constraints with the main variable x_2 are added to the set of unprocessed constraints U and removed from the set of processed constraints P. Because none of the constraints is processed there are no changes. Likewise, there is no stored information for x_2. The variable x_j is not assigned in S so there is nothing to delete in Line 20. (UNSAT, $\{I_1\}$) is returned in Line 21.

Continue in `get_unsat_cover` for x_1:
Because UNSAT was received `construct_characterization`$((S = x_0 \mapsto -1, x_1 \mapsto -1), \{I_1\})$ as described in Algorithm 4 is called in Line 15. There, because $P_2 = \{c_1\}$ in I_1 `get_disc`(c_1), as described in Algorithm 2 is called in Line 6. disc(c_1) has not been calculated and is therefore not stored. In the call to `get_disc`, it is first checked whether the discriminant is stored. It is not stored so we calculate it as $d_1 := \text{disc}(c_1) = 1 - 2x_1 - 2x_2 + x_1^2 + x_2^2$ in Line 4, store d_1 in Line 5 for future use and return it in Line 6. We have $L = \emptyset$ and $U = \emptyset$ in I_1 so there are no resultants to calculate in Line 8 and 9. The characterization $R := d_1$ is returned. Back in the main loop, the assignment for x_1 is removed from S and `interval_from_characterization`$(S = (x_0 \mapsto -1), x_1 \mapsto -1, R)$ in Line 17 is called and returns $\{I_2\}$ with :

$$I_2 : l := -\infty, u := \infty, L := \emptyset, U := \emptyset, P_2 := \{d_1\}, P_\perp := \emptyset \tag{3.2}$$

The algorithm for `interval_from_characterization` is given in [ÁDEK20] as algorithm 5. In Line 18 I_2 is added to $\mathbb{I}_{x_1}$, which now covers $\mathbb{R}$ and we exit the while-loop. There are no stored intervals for x_1 and x_1 is not assigned in S, therefore nothing is changed and (UNSAT, $\{I_2\}$) is returned.

Continue in `get_unsat_cover` for x_0:

Again, because UNSAT was received `construct_characterization`$((S = x_0 \mapsto -1), \{I_2\})$ is called. There, for the same reasons as above, only `get_disc`(d_1) is called. disc(d_1) is not stored and is therefore calculated as $d_2 := \text{disc}(d_1) = x_0^2 - 2x_1$, stored and returned. Thus, the characterization $R := \{d_2\}$ is returned. The assignment for x_0 is removed from S and `interval_from_characterization`$(S = (), x_0 \mapsto -1, R)$ is called and returns $\{I_3\}$ with :

$$I_3 : l := -\infty, u := 0, L := \emptyset, U := \{x_0\}, P_2 := \{x_1, x_1 - 2\}, P_\perp := \emptyset \tag{3.3}$$

In Line 18 I_3 is added to the (partial) cover. $\mathbb{I}_{x_0} = \{I_3\}$ does not cover $\mathbb{R}$ and we iterate through the main loop again. x_0 is not assigned in S and we sample $s_0 := x_0 \mapsto 1$ outside of $\mathbb{I}_{x_0}$ and have $S = (x_0 \mapsto 1)$. S does not have full dimension and we jump to the recursive call.

`get_unsat_cover`$(S = (x_0 \mapsto 1), x_1)$:

The call to `get_unsat_intervals` can not draw any conclusions as none of the unprocessed constraints have the main variable x_1 and therefore we set $\mathbb{I}_{x_1} := \emptyset$. There are no stored intervals for x_1 so there is nothing to load and we enter the while-loop. x_1 is not assigned in S so we sample $s_1 := x_0 \mapsto -1$ and set $S := (x_0 \mapsto -1, x_1 \mapsto -1)$. S does not have full dimension and we jump to the recursive call.

`get_unsat_cover`$(S = (x_0 \mapsto 1, x_1 \mapsto -1), x_2)$:

All unprocessed constraints have the main variable x_2 and have to be considered by the call to `get_unsat_intervals`. We receive the return value $\mathbb{I}_{x_2} := \{I_4\}$ with :

$$I_4 : l := -\infty, u := \infty, L := \emptyset, U := \emptyset, P_2 := \{c_1\}, P_\perp := \emptyset \tag{3.4}$$

$\mathbb{I}_{x_2}$ covers $\mathbb{R}$ and we therefore skip the while-loop and jump to Line 19. Because all constraints are unprocessed, there are no stored intervals for x_2 and x_2 is not assigned in S there are no changes by the call to `remove_dimension` and (UNSAT, $\{I_4\}$) is returned.

Continue in `get_unsat_cover` for x_1:

UNSAT was received and `construct_characterization`$((S = x_0 \mapsto 1, x_1 \mapsto -1), \{I_1\})$ is called. Because of $P_2 = \{c_1\}$ in I_4 `get_disc`(c_1) is called in Line 4. There, in Line 1 it is checked if the result is stored. This check is successful as disc(c_1) has been calculated before as d_1. Thus, the resulting polynomial d_1 is loaded from memory and returned in Line 2. The characterization $R := \{d_1\}$ is returned and and the assignment for x_1 is removed from S. Afterwards `interval_from_characterization`$(S = (x_0 \mapsto 1, x_1 \mapsto -1), R)$ is called and returns $\{I_5\}$ with :

$$I_5 : l := -\infty, u := 0, L := \emptyset, U := \{d_1\}, P_2 := \{d_1\}, P_\perp := \emptyset \tag{3.5}$$

$\mathbb{I}_{x_1} = \{I_5\}$ does not cover $\mathbb{R}$ and we iterate through the while-loop again. As x_1 is not assigned in S and we sample $s_1 := x_1 \mapsto 1$ outside of $\mathbb{I}_{x_1}$ and have $S = (x_0 \mapsto 1, x_1 \mapsto 1)$. S does not have full dimension and we jump to the recursive call.

`get_unsat_cover`$(S = (x_0 \mapsto 1, x_1 \mapsto 1), x_2)$:
All unprocessed constraints have the main variable x_2 and have to be considered by the call to `get_unsat_intervals`. We receive the intervals $\mathbb{I}_{x_2} := \{(-\infty, 0), [0,0], [2,2], (2,\infty)\}$. $\mathbb{I}_{x_2}$ does not cover $\mathbb{R}$ and we enter the while-loop. x_2 is not assigned in S and we sample $s_2 := x_2 \mapsto 1$ and set $S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 1)$. S has full dimension, i.e. have $j = 2 = n$ and the condition in Line 7 is fulfilled. We call Algorithm 5 in Line 8. In `store_dimension` the intervals $\mathbb{I}_{x_2}$ for x_2 are stored and the unprocessed constraints $\{c_1, c_2\}$ with main variable x_2 are added to the set of processed constraints P and removed from the set of unprocessed constraints U. We then have $P = \{c_1, c_2\}$ and $U = \emptyset$ and (SAT,S) is returned in Line 9.

Continue in `get_unsat_cover` for x_1:
SAT was returned by the recursive call and the if-condition in Line 11 is fulfilled. The information about x_1 is stored using a call to Algorithm 5 in Line 12. In `store_dimension` the intervals $\mathbb{I}_{x_1} = \{I_5\}$ get stored for x_1 There are no unprocessed constraints and (SAT,S) is passed on.

Continue in `get_unsat_cover` for x_0:
Likewise to above, SAT has been returned by the recursive call and the intervals $\mathbb{I}_{x_0} = \{I_3\}$ get stored for x_0. There are no unprocessed constraints and (SAT,S) is passed on.

Continue in `user_call`:
The initial call to `get_unsat_cover` returned SAT and we therefore enter the if-condition in Line 10. S is stored as a satisfying witness for future use in Line 11 and (SAT,S) is returned further. With that the CAC algorithm terminates.

SAT has been concluded with the satisfying witness $S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 1)$. The solver stored the intervals $\mathbb{I}_{x_0}$ for x_0, $\mathbb{I}_{x_1}$ for x_1, $\mathbb{I}_{x_2}$ for x_2 the satisfying witness S, the set of unprocessed constraints $U := \emptyset$ and the set of processed constraints $P := \{c_1, c_2\}$.

We now add a constraint and call the CAC algorithm again to present how SAT can be deducted in Algorithm 7. Consider that the following constraint is added and the CAC algorithm is called again.

- $c_3 := x_0^2 + x_1^2 - 2 \leq 0$

The surface of the polynomial on the left side of the constraint c_3 is presented in Figure 3.2. The constraint c_3 is given in blue. Newly added constraints are inserted in the set of unprocessed constraints and we set $U := \{c_3\}$ and jump to the `user_call`. The satisfying witness S, which was found in the previous call is saved and passed as an argument.

`user_call`$(S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 1))$:
In Line 1 the unprocessed constraints are evaluated at the current satisfying witness. We have $c_3(S) =$ True and is thus c_3 not added to F. Therefore we have $|F| = 0$ and (SAT, S) is returned in Line 3.

Again, SAT has been concluded with the satisfying witness $S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 1)$. The unprocessed constraints are not in conflict with S and therefore all

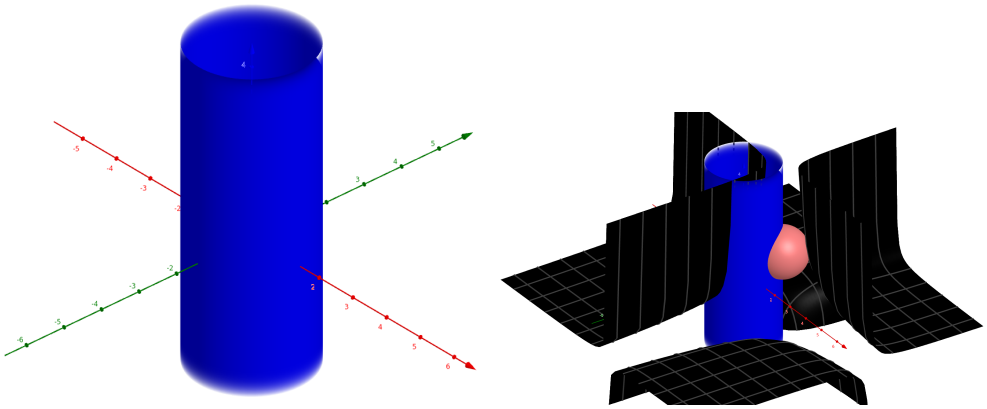

Figure 3.2: c_1 is red, c_2 is black and c_3 is presented in blue.

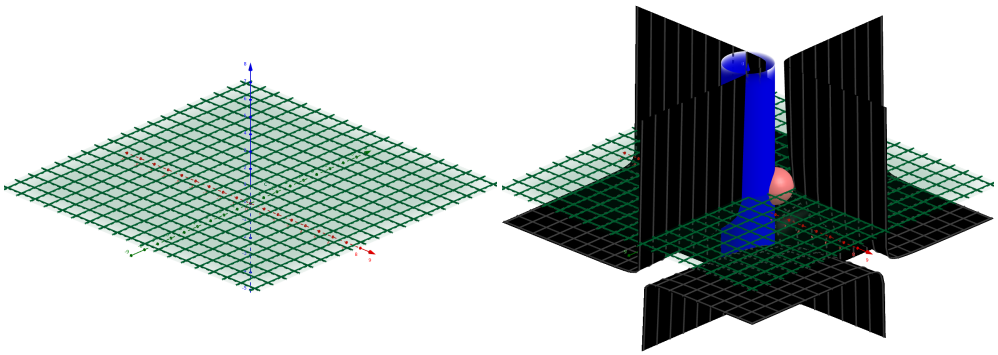

Figure 3.3: Visualization of the surfaces of the polynomials of the right side of the constraints in Section 3.3.

constraints are consistent. The stored intervals for x_0, x_1, x_2 and the satisfying witness S are kept unchanged. The set of unprocessed constraints $U := \{c_3\}$ and the set of processed constraints $P := \{c_1, c_2\}$ are stored. The constraint c_3 remains unprocessed. Note that the CAC algorithm terminated without calling `get_unsat_cover` and reevaluating the processed constraints.

We now add a constraint and call the CAC algorithm again to demonstrate how the saved intervals can be used. The following constraint is now added and the CAC algorithm is called again.

- $c_4 := x_2 - 1 < 0$

This constraint c_4 restricts the solution set to the extend that the old sample point is no longer a satisfying witness. Again, the surface of the polynomial of the left side of the constraint is visually presented in Figure 3.3. The green plane represents $c4$. In Figure 3.4 the surfaces of the polynomials are presented with fixed $x_0 = 1$.

Again, new constraints are added to the set of unprocessed constraints and we set $U := \{c_3, c_4\}$ and jump to the `user_call` and pass the saved satisfying witness S.

`user_call`$(S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto 1))$:

First, the unprocessed constraints $U = \{c_3, c_4\}$ are evaluated at S. We have

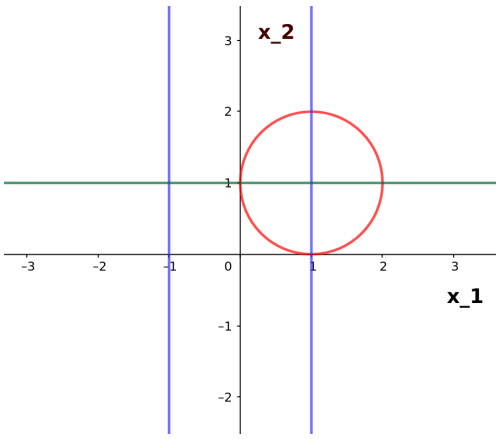

Figure 3.4: Visualization of the surfaces of the polynomials of the right side of the constraints in Section 3.3 with fixed $x_0 = 1$.

$c_3(S)$ = True and $c_4(S)$ = False. Therefore we set $F := \{c_4\}$ at Line 1. In the loop from Line 4 - 8 there is only the constraint c_4 to consider. The main variable is $\text{main}(c_4) = x_2$. In Line 6 the assignment for x_2 is removed from S so we now have $S = (x_0 \mapsto 1, x_1 \mapsto 1)$. We then iterate over the variables with a higher level. In this case only there are none and we skip the for-each loop.

`get_unsat_cover`$(S = (x_0 \mapsto 1, x_1 \mapsto 1), x_0)$:
As x_0 is assigned in S we call `get_unsat_intervals` for an empty sample point in Line 1 and consider all unprocessed constraints with the main variable x_0. There are none and accordingly the algorithm returns no intervals and we load the stored intervals for x_0 in Line 2. Thus, we have $\mathbb{I}_{x_0} := \{I_3\}$. Note that I_3 was defined in 3.3. $\mathbb{I}_{x_0}$ does not cover $\mathbb{R}$ and we go into the while-loop. x_0 is already assigned in S and S does not have full dimension so we jump into the recursive call.

`get_unsat_cover`$(S = (x_0 \mapsto 1, x_1 \mapsto 1), x_1)$:
Again, the current variable of interest is assigned in S and we call `get_unsat_intervals` for the sample point $(x_0 \mapsto 1)$. The unprocessed constraints c_3 has the main variable x_1 and has to be considered. The intervals $\mathbb{I}_{x_1} := \{(-\infty, -1), (1, \infty)\}$ are returned. Afterwards we load the stored intervals for x_1. The interval I_5, defined in 3.5 is loaded and we obtain $\mathbb{I}_{x_1} := \{(-\infty, -1), (-\infty, 0), (1, \infty)\}$. This does not cover $\mathbb{R}$ and go into the while-loop. x_1 is already assigned in S and S does not have full dimension so we jump into the recursive call.

`get_unsat_cover`$(S = (x_0 \mapsto 1, x_1 \mapsto 1), x_2)$:
x_2 is not assigned in S and `get_unsat_intervals` is called. There only c_4 has to be considered because it is the only unprocessed constraints with main variable x_2. $\mathbb{I}_{x_2} := \{[1, \infty)\}$ is returned. Afterwards in Line 2 the stored intervals for x_2 are loaded and we have the intervals $\mathbb{I}_{x_2} := \{(-\infty,0), [0,0], [1, \infty), [2,2], (2, \infty)\}$. $\mathbb{I}_{x_2}$ does not cover $\mathbb{R}$ and we enter the while-loop. x_2 is not assigned in S and we sample $s_2 := x_2 \mapsto \frac{1}{2}$ and set $S := (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto \frac{1}{2})$. S is now full dimensional and we enter the if-condition in Line 7. We update the stored

intervals for x_2 by storing $\mathbb{I}_{x_2}$ in Line 8 with a call to Algorithm 5. Further the unprocessed constraints with main variable x_2 are added to the set of processed constraints and removed from the set of unprocessed constraints so we set $U := \{c_3\}$ and $P := \{c_1, c_2, c_4\}$. (SAT, S) is returned.

Continue in `get_unsat_cover` for x_1:
SAT was returned by the recursive call and the if-condition in Line 11 is fulfilled. We update the stored intervals for x_1 by storing $\mathbb{I}_{x_1}$ in Line 12 with a call to Algorithm 5. Further the unprocessed constraints with main variable x_2 are added to the set of processed constraints and removed from the set of unprocessed constraints so we set $U := \emptyset$ and $P := \{c_1, c_2, c_3, c_4\}$. (SAT, S) is returned.

Continue in `get_unsat_cover` for x_0:
As above, SAT was returned by the recursive call and we update the stored intervals and the set of processed and unprocessed constraints. Since there are no added intervals and we have $U = \emptyset$ there are no changes and (SAT, S) is returned.

Continue in `user_call`:
The initial call to `get_unsat_cover` returned SAT. S is stored as a satisfying witness for future use in and (SAT,S) is returned further. With that the CAC algorithm terminates.

Again, SAT has been concluded with the satisfying witness $S = (x_0 \mapsto 1, x_1 \mapsto 1, x_2 \mapsto \frac{1}{2})$ and $U = \emptyset$ and $P = \{c_1, c_2, c_3, c_4\}$. In addition, the intervals for x_1, x_2, x_3 and some projections have been saved for possible further use.

Chapter 4

Implementation

Satisfiability-Modulo-Theories Real Arithmetic Toolbox (SMT-RAT) [CKJ+15] is an open-source toolbox for strategic and parallel SMT solving written in C++. The solver is maintained by the Theory of Hybrid Systems research group at the RWTH Aachen University. The program has a modular structure and consists of algorithms to solve quantifier-free (non-)linear real arithmetic. Essential for this thesis is that SMT-RAT is suitable for QF_NRA as described in 2.2 and includes an implementation of the cylindrical algebraic covering, as described in 2.4. The CAC algorithm was implemented according to the original paper [ÁDEK20] throughout a master thesis as the module *CADIntervalModule* [Fra20]. SMT-RAT uses *Computer ARithmetic Library* (CArL), an open-source C++ library for computer arithmetic and logic [CKJ+20]. CArL is maintained by the Theory of Hybrid Systems research group at the RWTH Aachen University. CArL implements algorithms to calculate the resultants and discriminants of polynomials, as described in Definition 2.4.6, SMT-RAT also provides other tools for debugging, preprocessing and benchmarking. *Benchmax* is a tool for automated benchmarking and gathering statistics, which are defined by the used module. It allows specifying a time and a memory limit for each execution of the solver. A module consists of several main interfaces :

- *addCore:* Takes constraints as parameter. The module has to take the received constraint into account.

- *removeCore:* Takes constraints as parameter. Removes the received constraints from the module. This includes all calculations made on the basis of these constraints.

- *checkCore:* Checks the received set of constraints for consistency.

The changes described in Chapter 3 were implemented based on the non-incremental implementation of the CAC algorithm in the module *CADIntervalModule* by [Fra20]. Accordingly, we present not the complete implementation of the CAC algorithm, but only the changes made to the already existing implementation. The module contains a class for the CAC algorithm, in which the actual implementation is located.

4.1 Projection Memory

In the following section, the implementation of the projection memory, as presented in Section 3.1, is described. For that, the implementation of `construct_characterization` has to be modified. We exchange the call to calculate the resultant with a call to a function representing Algorithm 3 and the call to calculate the discriminant with a call to a function representing Algorithm 2. To implement these functions, we need a suitable data structure to store the result and parameters. This data structure should provide fast insertion, lookup and access times of elements that consist of both the polynomial (pair of polynomials) from which the discriminant (the resultant) will be calculated and the corresponding resulting polynomial. The C++ standard [ISO12] provides two data structures that are well suited to the requirements needed. The `ordered map`, which is (often) implemented using a red-black tree and the `unordered map`, which is (often) implemented using a hash table [CLRS09]. In general, maps are associative containers that store elements formed by a combination of a *key-value* and a *mapped-value* that supports insertion and removal of elements but does not provide a way to insert an element at a specific position. The key-value uniquely identifies the mapped-value stored as content. The mapped-value can only be accessed using the corresponding key-value. In a map the data-type used for the key-values must be the same for all elements. Likewise, the data-type used for the mapped-values must be the same for all elements. However, the data-type of the key-values and the mapped-values can be different. When storing a discriminant, the key-value is a single polynomial p and the mapped-value is $\mathrm{disc}(p)$. When storing a resultant, the key-value is a pair of polynomials (p,q) and the mapped-value is $\mathrm{res}(p,q)$. We use two different maps to store the discriminant and the resultant because the key-value is different.

In an `ordered map`, elements are stored following a given order of the key-values. CArL provides a strict partial ordering $<$ on polynomials. The strict partial ordering $<$ for polynomials is implemented in CArL. When storing a discriminant, we have a single polynomial as the key-value. Thus, we use the $<$ operator for polynomials to provide the necessary ordering. However, when storing a resultant, we have a pair of polynomials as the key-value and $<$ can not be used directly. We now extend the $<$ operator to work for pairs of polynomials. We first consider the respective first polynomials and compare them with the $<$ operator. If these are equivalent, we compare the second polynomials in the pair with the $<$ operator. To ensure that the order of polynomials in the pairs does not influence on the mapped-value as presented in Algorithm 3, we put the smaller polynomial in the $<$ order for polynomials in the first place of the pair. Using this, when comparing two pairs of polynomials the outcome stays the same, even when the polynomials in the pairs change position. This corresponds to the pseudo-code presented in Algorithm 9. We use this extended $<$ operator for pairs of polynomials to order the key-values of resultants.

In an `unordered map`, the elements are not ordered in any particular way but are organized into *buckets*. The bucket into which an item is placed depends entirely on the *hash-value* of the key-value. A hash function for single polynomials is provided in CArL. Thus, when placing elements representing discriminant into the corresponding bucket, we use this hash function to calculate the hash-value of the key-value. However, this does not work when storing resultants as the key-value is a pair of polynomials. The C++ standard does not provide a way to calculate the hash-value of pairs. To overcome this, we first calculate the hash-value of the individ-

Algorithm 9: $<$ operator for pairs of polynomials.

Data: Set of Processed Constraints P, Set of Unprocessed Constraints U
Input: Two pairs of polynomials (a,b),(p,q) with $a < b$ and $p < q$.
Output: True of $(a,b) < (p,q)$, False otherwise.

```
1 if a = q then
2 |  return b < q
3 else
4 |  return a < p
```

	`Ordered Map`	`Unordered Map`
Insert:	$\mathcal{O}(\log(N))$	Average: $\mathcal{O}(1)$ Worst Case: $\mathcal{O}(N)$
Access:	$\mathcal{O}(\log(N))$	Average: $\mathcal{O}(1)$ Worst Case: $\mathcal{O}(N)$
Check existence:	$\mathcal{O}(\log(N))$	Average: $\mathcal{O}(1)$ Worst Case: $\mathcal{O}(N)$

Table 4.1: Complexity of insertion, access and checking for existence of an element in an `ordered map` and in an `unordered map`. Let N be the amount of elements stored.

ual polynomials in the pair and use the *hash_ combine* function provided by the boost library [Sch11], to combine the two hash-values into a single one. The output of the hash_combine function changes when the order of the parameters changes. To ensure that the order of polynomials in the pair does not influence the pair's hash-value, we put the smaller polynomial in the $<$ ordering for polynomials in the first place of the pair. This way, pairs of polynomials have the same hash-value even if the polynomials change position in the pair.

The complexity of the operations in the `ordered map` and the `unordered map` is presented in Table 4.1. However, these do not allow any statement about which of the two types of maps is more suitable. The `unordered map` is heavily dependent on the runtime and quality of the used hash function. If a different hash function is implemented in CArL, the effective runtime of the operations in the `unordered map` also changes. It remains to be said that a problem of using the `unordered map` is that different polynomials could have the same hash-value. This could lead to an incorrect result being returned for these polynomials which could lead to the algorithm no longer being correct. However, this is a statistically rare occasion and did not occur in the benchmarks presetned in Chapter 5.

4.2 Incrementality

In the following, the implementation of the incremental approach, as presented in Section 3.2, is described. We first expand the module so that it stores an object for the sample point. When the sample point is passed to a function, we always

pass a reference to that object. Thus, the sample point is stored globally. If SAT was concluded for a set of constraints, this object stores a satisfying witness and if UNSAT was concluded, this object is empty. As given in Definition 3.2.1, we extend the module to store an array for the set of processed constraints and an array for the set of unprocessed constraints. We further expand the module to store the regions of unsatisfiability of the processed constraints. This is done using an `ordered map`, which stores elements formed by a combination of a key-value and a mapped-value [ISO12]. The key-values uniquely identify the mapped-values stored as content. The key-value of an element is a variable and the corresponding mapped-value is the set of intervals that have been concluded for that variable. The data type of the mapped-value is an set of intervals, as defined in Definition 2.4.5 and the data type key-value is variable. The mapped-value for the corresponding variable is the set of intervals that have been concluded in that dimension in a previous call of the CAC algorithm. I.e., to store the set of intervals $\mathbb{I}_{x_1}$ that have been concluded for x_1, we use x_1 as the key-value and $\mathbb{I}_{x_1}$ as the mapped-value. The key-values are ordered according to the $<$ operation on integers and the corresponding variable level. To store intervals for a dimension, we add an element to the map, using the corresponding variable as the key-value and the intervals as the mapped-value. To remove the intervals for a dimension, we remove the corresponding element from the map.

When *addCore* is called to add constraints to the module, we append the passed constraints to the array of unprocessed constraints. When constraints are removed from the CAC module, the stored information about each of the variables is removed and all constraints are removed from the array of processed constraints and added to the array of unprocessed constraints. When the *removeCore* function is called to remove constraints, we first remove the passed constraints either from the array of processed constraints or the array of unprocessed constraints, remove all stored intervals and clear the currently saved sample point. Further, we append the array of processed constraints to the array of unprocessed constraints and subsequently clear the array of unprocessed constraints. Thus for the next call to *checkCore*, all constraints are unprocessed, the stored sample point is empty and the map of intervals does not contain any information about the variables.

The *checkCore* function represents the `user_call`. We will now present how the `user_call` is implemented as presented in Algorithm 7. We first check whether the passed sample point has full dimension. If not, the previous call to the module (if there was any) did not return SAT or constraints have been removed. Thus there are no stored intervals for the variables and that all constraints are unprocessed and we can skip to the call of `get_unsat_cover` first variable in the variable ordering. If the sample point has full dimension, the previous call to the module concluded SAT stored the satisfying witness. In this case, we first evaluate all unprocessed constraints at the passed sample point and add constraints that do not evaluate to true to a new array. CArL provides a method to evaluate constraints at a given sample point. If all unprocessed constraints evaluate to True, the old sample point is still a satisfying assignment and we can conclude SAT and return the stored sample point. Otherwise, the variable that places the lowest in the variable ordering and is the main variable of one of the unprocessed constraints that did not evaluate to true under the sample point is determined. We remove the variable assignment for this variable in the sample point. We then iterate over all variables that are higher in the ordering and remove the assignment and the stored intervals. Additionally, we remove all constraints with a higher level from the array of processed constraints and

add them to the set of unprocessed constraints. This is done by looping over the variables of each processed constraint and checking whether one of those variables is higher in the variable ordering. In this case we remove that constraint from the array of processed constraints and add it to the array of unprocessed constraints. Afterwards `get_unsat_cover` we call for the first variable in the ordering. Based on the output of `get_unsat_cover`, either SAT or UNSAT is returned. If UNSAT has been concluded, we clear the sample point stored by the module.

Now the incremental implementation of the `get_unsat_cover`, as described in Algorithm 8 is presented. One difference to the existing implementation is that the variable currently considered may already be assigned in the sample point. If that is the case, all unprocessed constraints with that main variable must evaluate to true when evaluated at the passed sample point. Thus, if the variable is already assigned, we can skip the call to `get_unsat_intervals` as no more unsatisfiable intervals can be concluded. Thus, if the variable is already assigned, we create a copy of the sample point and remove the assignment for the current variable and all variables of an higher level. Using this copy of the sample point `get_unsat_intervals` to get all intervals in which any unprocessed constraints with the current variable as the main variable is unsatisfied. If the variable is not assigned, we call `get_unsat_intervals` with the original sample point. Further, we load the stored unsatisfying intervals for the current dimension's processed constraints if there are any. This is done by accessing the stored intervals using the currently considered variable as the key-value. In addition, the resampling of the current variable in the while-loop has the be changed as the currently considered variable might already be assigned. We do nothing if the current variable is already sampled and jump to the recursive call to explore the next dimension. Otherwise, we choose a point with a call to `sample_outside` and use it as an assignment for the current dimension's variable. This assignment is added to the sample point. When the sample point has full dimension or SAT is received by the recursive call, the sample point is a satisfying witness. Thus we store the concluded intervals for the corresponding variable. We overwrite the stored intervals if there are any. This way, no intervals are lost or saved multiple times because the stored intervals are contained in the new set of intervals. When storing intervals for a variable, we always set the unprocessed constraints with that main variable to be processed. These constraints are removed from the array of unprocessed constraints and added to the array of processed constraints. This is done by iterating over the variables of each unprocessed constraint and checking whether the current variable of interest is used and no other variable that is higher in the variable ordering. If UNSAT was received by the recursive call, the assignment in the sample point for the currently considered variable is removed. This allows the variable to be sampled again in the next iteration of the while-loop. When the unsatisfying intervals form a cover for $\mathbb{R}$ the while-loop is exited. Conclusively, the processed constraints with the main variable that is currently being considered are set to be unprocessed. This is done by iterating over the variables of the individual processed constraints and checking whether the currently considered variable is existent and no other that is higher in the variable ordering. Further, we remove the assignment for the currently considered variable from the sample point and remove the stored intervals.

Chapter 5

Test Results

In the following chapter we present how the modified CAC algorithm performs in terms of runtime and memory usage. First the changes are considered individually. We consider the projection memory in Section 5.1 and the incrementality in Section 5.2. Then we look at both changes together in Section 5.3. To evaluate the the changes in a practical context, we use the benchmark files QF_NRA from SMT-LIB, which consists of 10 Folders with various amounts of files and complexity [BFT16]. The benchmark consists of 11489 files which vary in complexity. The data stems from runs on AMD Opteron 6172 processors, with a total of 4GB RAM per file. The time limit for each file is 2 minutes. Files that exceed the maximum runtime are listed as *timeout* and files that exceed the maximum memory usage are listed as *memout*. Files that returned either "SAT" or "UNSAT" are listed as *solved*. In order to evaluate the improvements, we use the implementation of the unmodified CAC algorithm, which is described in [Fra20]. This unmodified version of the CAC algorithm does not use a data structure to store projections and works non-incremental. The unmodified version of the algorithm, as well as all modified version were implemented as a module in the SMT-RAT solver, which is presented in detail in [CKJ+15].

5.1 Projection Memory

In the following we present the effective runtimes of the insertion, access and existence check of elements for both types of maps used in the implementation of the projection memory. The presented times stem from an Intel i5-8250U with 8GB RAM running Debian 10. Each measurement was carried out ten times and the mean running time is displayed. We randomly generate polynomials for the key-value and mapped-value for measuring the insertion time of discriminant calculations. We create pairs of randomly generated polynomials for the key-value to measure the insertion time of resultant calculations. For the mapped-value, we again use a randomly generated polynomial. For $1 \leq i \leq 100$ we generate a given amount of polynomials with i variables. The algorithm that generates these random polynomials is implemented in CArL [CKJ+20]. The set of polynomials inserted into the ordered map and the unordered map is identical. The results are presented in Table 5.1. To measure the time to check for an elements' existence, we first insert 10^4 randomly generated polynomials with less than 100 variables. We then measure the time it takes to check for the existence of a given amount of elements. Half of the elements checked

Discriminant			Resultant		
Amount per level	Ordered Map	Unordered Map	Amount per level	Ordered Map	Unordered Map
1	1ms	1ms	1	2ms	1ms
50	82.3ms	78.6ms	50	117.9ms	130.0ms
100	170.7ms	164.7ms	100	247.1ms	274.9ms
500	876.6ms	984.5ms	500	1341.0ms	1548.8ms
1000	1776.4ms	2301.9ms	1000	2763.4ms	3480.4ms

Table 5.1: Insertion: Randomly generated polynomials with fewer than 100 variables are used as key value and mapped value.

are present have been inserted into the map and the other half consists of newly generated polynomials. We insert the same set of polynomials in both the ordered and the unordered map. The set of elements we check the existence of is identical for the ordered map and the unordered map. The results are presented in Table 5.2. To

Discriminant			Resultant		
Amount	Ordered Map	Unordered Map	Amount	Ordered Map	Unordered Map
10^2	3ms	4ms	10^2	7.0ms	14.0ms
10^3	41.2ms	54.3ms	10^3	77.2ms	139.7ms
10^4	418.1ms	545.7ms	10^4	777.4ms	545.7ms
10^5	4171.9ms	5452.9ms	10^5	7742.2ms	5452.9ms

Table 5.2: Check Existence of polynomials in maps with 10^4 inserted elements.

measure an element's access time, we first insert 10^4 randomly generated polynomials with less than 100 variables. We then measure the time it takes to access a given amount of polynomials present in the map. We insert the same set of polynomials in both the ordered and the unordered map. The set of elements we access is identical for the ordered map and the unordered map. The results are presented in Table 5.3. The runtimes of the operations do not provide any conclusion about whether the ordered map or the unordered map are more efficient. Going forward, we will use the ordered map as the underlying data structure for the projection memory. However, it can be said that the operations each have a short runtime, and therefore the threshold for

Discriminant		
Amount	Ordered Map	Unordered Map
10^2	1ms	14.0ms
10^3	16.2ms	139.7ms
10^4	154.5ms	545.7ms
10^5	1532.ms	5452.9ms

Resultant		
Amount	Ordered Map	Unordered Map
10^2	3ms	4ms
10^3	41.2ms	54.3ms
10^4	418.1ms	545.7ms
10^5	4171.9ms	5452.9ms

Table 5.3: Access of polynomials in maps with 10^4 inserted elements.

the minimum runtime of the calculations that are to be saved should also be low.

Heuristics can be used to filter what elements are inserted or to delete elements to prevent the data structures from becoming large, resulting in long lookup and access times. As a heuristic for insertion, we measure the time it takes to calculate the resultant or discriminant. If the calculation exceeds a given *threshold*, we insert the corresponding element into the data structure. If this threshold is not exceeded, we do not insert the element.

In Table 5.4, we present how many projections calculations can be saved given a certain threshold of minimum calculation time for insertion of elements. For every combination of folder and threshold, the cell contains two numbers. The top one is the number of resultant and discriminant and resultant calculations without the presented data structures. The bottom one is the number of resultant and discriminant calculations with the presented data structures given the various thresholds. The difference between the top and bottom numbers is the amount of calculations avoided by loading the result from the corresponding data structures. Table 5.4 shows that the majority of all calculated projections have a calculation time of less than 100ms. Of all calculated projections, only 0.087% have a calculation time of more than 100ms and only 0.0039% of a calculated projections have a runtime of more than 2000ms. Additionally it is presented, that 25.54% of all projection calculations can be avoided with a projection memory and a threshold of 0ms.

We now show how many files of the QF_NRA benchmark can be solved with the projection memory compared to the amount of files solved without the projection memory.

In Table 5.5, we show how the implementation using the projection memory, compares to the unmodified version. The minimum calculation time required for insertion of the projection is given as the threshold. The number of files that are solved and the number of memouts and timeouts are listed. Additionally, the average runtime in seconds, and the average memory usage in bytes of the solved files is listed. In all cases, using the projection memory more files could be solved and hence a performance boost is achieved compared to the unmodified implementation. Using a threshold of 0ms all projections are inserted. Using this approach 8431 files were solved, 5 more compared to the implementation not using the projection memory. But this also has

Folder \ Threshold	0ms	100ms	500ms	1000ms	2000ms	5000ms
Sturm	10465	176	98	80	32	31
414 Files	8933	34	17	15	8	7
InvariantSynthesis	1504404	1608	95	23	0	0
69 Files	1311028	1582	94	22	0	0
Economics-Mulligan	572529	623	220	102	21	21
135 Files	378644	93	45	30	11	11
hong	635370	419	71	55	6	6
20 Files	426871	10	5	4	1	1
hycomp	5768226	5480	1514	870	252	243
2752 Files	4599615	3269	672	400	147	140
kissing	29329	297	122	109	92	89
45 Files	15936	256	105	94	79	75
LassoRanker	2121111	2042	441	164	37	35
821 Files	1511130	1992	407	154	35	33
meti-tarski	180418	1096	660	334	128	128
7006 Files	138645	294	196	92	55	55
UltimateAutomizer	2425635	125	57	35	10	5
61 Files	1543055	122	55	33	8	4
zankl	2304292	1762	322	171	43	43
166 Files	1645885	1701	289	153	39	39
Total	15551779	13628	3600	1943	617	601
11489 Files	11579742	9355	1885	997	376	365

Table 5.4: Total amount and distinct amount of resultant and discriminant calculations with various minimum computation time. Each file has a timeout of 2 minutes

Threshold	Solved	Memout	Timeout	Avg. Runtime	Avg. Memory
0ms	8431	299	2759	1.38	62295
50ms	8427	297	2764	1.36	62287
200ms	8427	297	2764	1.35	62284
500ms	8426	296	2767	1.35	62251
No Memory	8424	297	2768	1.36	61980

Table 5.5: Comparison of the projection memory using different minimum calculation times as the threshold.

Approach	Solved	Memout	Timeout
Incremental	8455	306	2728
Non-Incremental	8424	297	2768

Table 5.6: Comparison of the incremental implementation and the non-incremental implementation.

an effect on memory usage, 299 files reached the memory limit, 3 more compared to the implementation not using the projection memory. With a higher threshold, less memory is used, but fewer files could be solved. Using a threshold of 50ms, 8427 files could be solved, 4 less than with a threshold of 0ms. However, 297 files reached the maximum memory usage, 2 less than with a memory threshold. Although the different thresholds only result in a marginal change of average runtime,files solved and memory, we can conclude that the implementation using a threshold of 0ms results in the largest increase of performance.

5.2 Incrementality

In Table 5.6, we present how the incremental approach without projection memory performs in comparison with the non-incremental approach. The incremental approach is able to solve 8455 files, which is an increase of 0.36% compared to the non-incremental approach. Comparing to the implementation using just the projection memory, the implementation using the incremental approach is able to solve more files, thus providing a larger performance increase. Using the incremental approach, 306 files reach the memory limit, which is an increase of 2.94% compared with

Approach	Solved	Memout	Timeout
Incremental 0ms	8491	306	2692
Incremental 50ms	8486	306	2697

Table 5.7: Runtime of the solved files using the non-incremental approach without projection memory and the incremental approach with projection memory and a threshold of 0ms and 50ms.

the non-incremental approach. This increase in memouts could be due to the data structures used in the incremental approach, or because the files, that reached the time limit using the non-incremental approach, have progressed progressed further and more information have been concluded and stored. In fact, all but one additional memout for the incremental implementations are timeouts for non-incremental implementation. Thus, the addition of incrementality to the implementation results in a performance increase and more memory usage compared to the non-incremental approach.

5.3 Incrementality and Projection Memory

In the following, we evaluate the incremental implementation which uses the projection memory. We use a threshold of 0ms and 50ms for insertion in the projection memory, as this results in the biggest performance increase. In Figure 5.1 we present the runtimes of all solved files using the non-incremental approach without projection memory and the incremental approach with projection memory and a threshold of 0ms and 50ms. In Table 5.7, it is shown how many files were solved, reached the memory limit or the maximum computing time. Using the incremental approach and a projection memory using a threshold of 0ms for insertion 8491 file were solved, which is 0.8% more compared to the original implementation. Using a threshold of 50ms, slightly less files could be solved with 8486. In Figure 5.1, we present the runtimes of the solved files. It can be seen that many files are solved with very little computing time. These files do not offer the possibility to be solved notably faster, as they are solved either directly using the Boolean structure or are trivial problems which are either solved without any projections or multiple calls to the theory solver. In Table 5.8, we present how many files can be solved without any theory call, without any incremental theory call, without any projections, or in less than 100ms in general. An incremental theory call is a theory call that can reuse the intervals concluded in the previous one. To get a better overview of the runtimes of files which are more complex, and therefore have potential for time savings, files with a minimum solving time of 1000ms have to be considered. To do this, we use the same data as in Figure 5.1, but filter out all files that have been solved in less than 1000ms by each of the implementations. This is is presented in Figure 5.2. It can be seen that the incremental approach solves more involved problems in a shorter time, thus resulting in more problems solved overall.

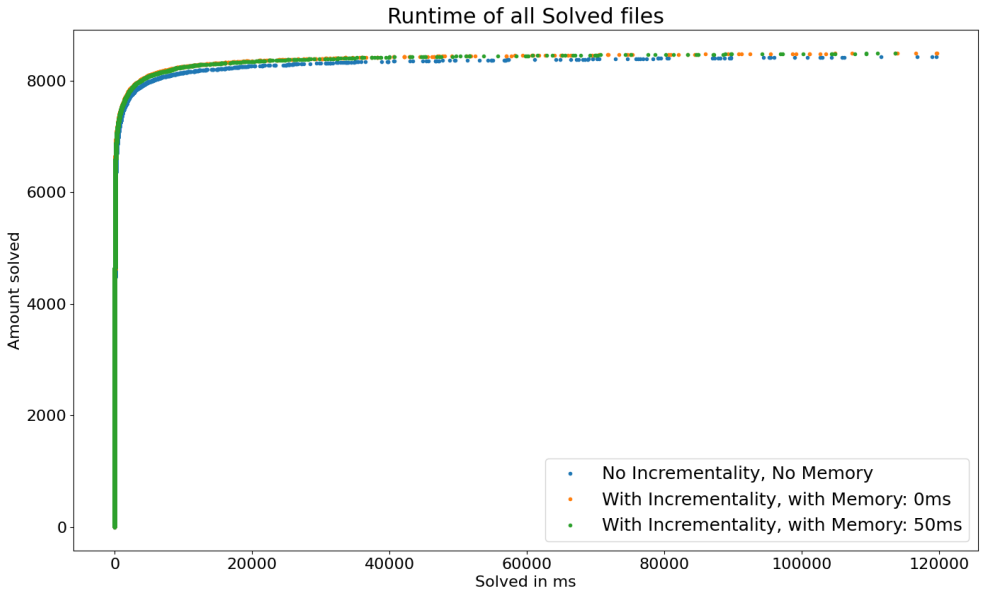

Figure 5.1: Runtime of the solved files for the non-incremental approach without projection memory and the incremental approach with projection memory using a threshold of 0ms and 50ms.

Solved without theory call	Solved without incremental theory call	Solved without any projections	Solved in less than 1000ms
1990	6917	3842	7408

Table 5.8: Number of files in the QF_NRA benchmark than can be solved with with the given properties.

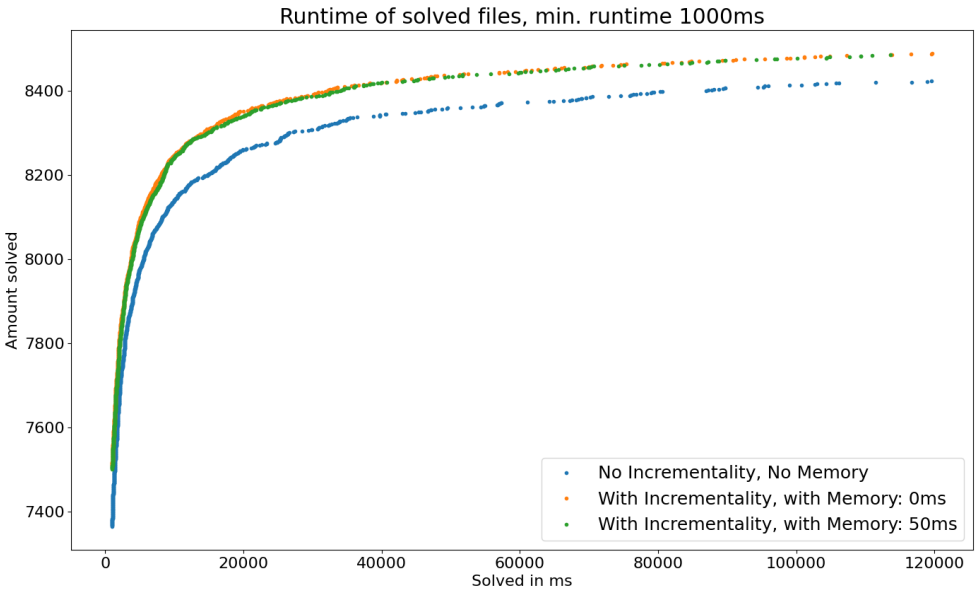

Figure 5.2: Runtime of the solved files with a minimum computation time of 1000ms for the non-incremental approach without projection memory and the incremental approach with projection memory using a threshold of 0ms and 50ms.

At the end of this chapter, we come to the conclusion that both the projection memory and the incrementality result in an increase of performance and an increase memory usage of the implementation of the CAC algorithm. The different thresholds used for the minimum calculation time of projections which are inserted into the projection memory result in only a marginal change. The addition of the projection memory in total results in a relatively small increase in performance compared to the addition of incrementality. The combination of the two additions brings the greatest performance improvement for implementation of the CAC algorithm. But we also come to the conclusion that the used QF_NRA benchmarks (SMT-LIB) [ISO12] may not be well suited as a rather large portion can be solved in a comparatively short time or without the need for a theory solver at all. This does not provide any room for noticeable improvements in the computation times.

Chapter 6

Conclusion

In this thesis, two improvements to the Cylindrical Algebraic Covering algorithm, described in the paper by Ábrahám, Davenport, England, and Kremer [ÁDEK20], are presented and implemented. A data structure is presented to store the results of calculations on polynomials, such that they can be reused at a later time. Two different approaches for storing the polynomials are considered. An ordered map, which works based on comparing polynomials or pairs of polynomials, and an unordered map, which works by storing the hash-values of the corresponding polynomials or pairs of polynomials. Heuristics can be used to determine which calculations are worth saving. We measure the time needed to calculate the result and store it if the time exceeds a certain threshold.

Additionally, it is presented how the algorithm can be adapted to work in an incremental way. To enable this, the constraints are split into processed constraints and unprocessed constraints. No information is known about the unprocessed constraints, while the unsatisfiable intervals of the processed constraints are known and stored. During the runtime of the algorithm, only the unsatisfiable intervals of the unprocessed constraints must be concluded, as the intervals for the processed constraints is loaded from memory. When all information about an unprocessed constraint is known, it becomes a processed constraint. When the sample point changes or constraints are removed, the corresponding information is deleted and the respective constraints are unprocessed again.

These changes were implemented with the idea of improving running time. Both changes bring a performance improvement, but also result in more memory usage. The implementation of the ordered map and the unordered map perform equally well. Different thresholds for insertion in the ordered map and the unordered map implementation perform alike, and we conclude that it benefits the performance most if all calculation is saved. However, these improvements in performance are minor. The addition of incrementality brings a comparatively larger increase in performance. The changes can also be used simultaneously, which results in the greatest increase of performance. Using both changes 0.8% more files if the QF_NRA benchmark could be solved compared to the additional implementation.

In the future, the algorithm could be expanded to include *backtracking*. Currently, if a constraint is removed from the solver, the stored information about all constraints is removed. Deleting just the information about the removed constraints would result another increase in performance.

Bibliography

[ÁDEK20] 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. *CoRR*, abs/2003.05633, 2020.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BPR06] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry (Algorithms and Computation in Mathematics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[CKJ+15] 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.

[CKJ+20] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Computer ARithmetic Library CArL, October 2020. `https://github.com/smtrat/carl`.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Brakhage, editor, *Automata Theory and Formal Languages*, pages 134–183, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[Duc00] Lionel Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145(2):149 – 163, 2000.

[Fra20] Hannah Franzen. Conflict Driven Cylindrical Algebraic Coverings for Nonlinear Arithmetic in SMT Solving. RWTH Aachen, Theory of Hybrid Systems, 2020.

[HBA+20] M. Hohenwarter, M. Borcherds, G. Ancsin, B. Bencze, M. Blossier, A. Delobelle, C. Denizet, J. Éliás, Á Fekete, L. Gál, Z. Konečný, Z. Kovács, S. Lizelfelner, B. Parisse, and G. Sturr. GeoGebra 6, October 2020. `http://www.geogebra.org`.

[ISO12] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, February 2012.

[Jir95] Mats Jirstrand. Cylindrical algebraic decomposition - an introduction. Technical Report 1807, Linköping University, Automatic Control, 1995.

[Sch11] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[Tar98] 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, Vienna, 1998. Springer Vienna.