

MASTER OF SCIENCE THESIS

INTERVAL CONSTRAINT
PROPAGATION IN
SMT COMPLIANT DECISION
PROCEDURES

Stefan Schupp

Supervisors:

Prof. Dr. Erika Ábrahám

Prof. Dr. Peter Rossmanith

Advisors:

Dipl.-Inform. Ulrich Loup

Dipl.-Inform. Florian Corzilius

March 2013

Abstract

There is a wide range of decision procedures available for solving the existential fragment of first order theory of linear real algebra (QFLRA). However, for formulas of the theory of quantifier-free nonlinear real arithmetic (QFNRA), which are much harder to solve, there are only few decision procedures (the lower bound for complete solvers is exponential). The context this thesis is settled in is the software project SMT-RAT, a software framework for SAT Modulo Theories (SMT) solving. SMT solving is a combination of a SAT solver, which checks the Boolean skeleton of a given input formula and a theory solver, which handles the involved theory constraints. SMT-RAT maintains different complete and incomplete solving modules and allows to combine several modules to operate as a theory solver.

Interval constraint propagation (ICP) is an incomplete decision procedure to efficiently reduce the domain of a set of variables with respect to a conjunction of polynomial constraints. The goal of this thesis is to present a module based on ICP for SMT-RAT. This module takes a conjunction of polynomial constraints as well as an initial set of boundaries, represented by intervals, for the variables occurring in the constraints as an input. It utilizes an interval extension of Newton's method to reduce the domain of the given variables up to a certain bound and passes the set of constraints as well as the new boundaries to complete solvers, such as the module based on the cylindrical algebraic decomposition (CAD).

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.

Stefan Schupp
Aachen, den 30. März 2013

Acknowledgements

I would like to express my gratitude to Prof. Dr. Erika Ábrahám, which gave me the opportunity to write this thesis and supported me throughout the creation of this thesis. Furthermore, I want to thank Dr. Prof. Peter Rossmann for offering his services as my second supervisor.

I also would like to show my gratitude to my advisers Florian and Ulrich – without your effort and support this thesis would not have been possible. I learned a lot from you during the past months, which helped me to develop my skills in both, theoretical and practical matters.

Last, but not least I thank my family, my flat-mates and my friends, which supported and encouraged me, especially during the last period of my thesis.

Contents

1	Introduction	9
2	Interval arithmetic	11
2.1	Basic operations	11
3	SMT solving	17
3.1	QFNRA formulae	17
3.2	Bounded NRA	18
3.3	Classic SMT solving	19
3.4	SMT-RAT	20
4	The ICP module	23
4.1	Algorithm	23
4.2	Preprocessing	24
4.3	Consistency check	26
4.4	Contraction/ICP	27
4.5	Split	34
4.6	Validation	37
4.7	Incrementality	41
5	Conclusion	45
5.1	Results	45
5.2	Future work	48
5.3	Summary	49
	Bibliography	51

Chapter 1

Introduction

Satisfiability checking has become more and more important during the last decades. The *SAT Modulo Theories* (SMT) approach is widely used for decision procedures, for example for the analysis of physical or chemical processes or real-time systems. The approach combines the usage of highly efficient *SAT solvers* with dedicated *theory solvers*.

In this thesis we focus on the *quantifier-free nonlinear real arithmetic* (QFNRA), which is convenient to describe the above mentioned problems. Currently there exist only few approaches for the satisfiability check of QFNRA formulae as they are hard to solve. Complete methods for QFNRA formulae have a lower complexity bound which is exponential. Among others, there exists the approach of the *cylindrical algebraic decomposition* (CAD) [Col74] which is complete. *Virtual substitution* (VS) [Wei88] and *Gröbner bases* [BKW93] are both incomplete procedures for QFNRA formulae. The context this thesis is settled in is the project SMT-RAT [CA11] with the aim to provide open-source software modules, which can be used for the development of SMT solvers. This toolbox includes various modules, among them modules which implement Gröbner bases, VS or CAD. Other modules currently implemented handle the conversion of input formulae to conjunctive normal form (CNF), linear real algebra (LRA) solving [Dan98, DM06] or the preprocessing of formulae. Additionally, SMT-RAT provides a manager where the single modules can be combined to a solving strategy.

The goal of this thesis is to develop a module for SMT-RAT which allows to efficiently reduce intervals, which form over-approximations of the solution space of variables contained in NRA formulae, by implementing an *interval constraint propagation* (ICP) [VHMK97] algorithm. This branch and prune algorithm is used to reduce the search space by contraction of interval domains for the variables of the input formula.

The implementation as an SMT-RAT module poses several difficulties: The module has to be integrable and thus fulfill certain constraints such as incrementality and compatibility to the demanded structure and communication interfaces of an SMT-RAT module. Furthermore, the module should be as independent as possible, which also includes an internal handling of solver states and an own preprocessing.

So far there exist implementations of the ICP algorithm, such as RealPaver [GB06], which implements ICP standalone and iSAT [FHT⁺07], which combines

ICP with SAT. This thesis is based on the proposal of Gao et al. [GGI⁺10], which combines ICP and LRA methods. We use this approach as a basis for our own enhancements based on the work of Goualard et al. [GJ08] towards reinforced learning and Herbort et al. [HR97] concerning the Newton-operator.

First we will provide background knowledge on interval arithmetic in Chapter 2 which includes definitions and notations as well as operations needed for the ICP algorithm.

After that in Chapter 3 we will focus on SMT solving and especially on the toolbox SMT-RAT in detail. There we will also sketch the structure and nature of the input formulae and then focus on the communication between the different modules of SMT-RAT presenting the essence of the most important functions. The main Chapter 4 is about the ICP algorithm. It consists of two parts – on the one hand the ICP algorithm itself and on the other hand the description of the implementation in the ICP module. The technical details on the single parts of the algorithm are presented as well as the adaption to embed the designed module in SMT-RAT.

Results from a simple example and an outlook as well as a summary conclude this thesis in Chapter 5.

Chapter 2

Interval arithmetic

In this section we present basic operations of interval arithmetic as defined in [Kul08]. The ICP module operates on multivariate polynomials defined over interval domains (see Chapter 4). As ICP only uses the basic calculations addition, subtraction, multiplication and division, it suffices to extend only those operations to intervals.

There is more than one way to extend arithmetic operations to intervals (e.g., the natural interval extension, the distributed interval extension or the Taylor interval extension, see [VHMK97]). We only refer to the natural interval extension as it is the most intuitive one and parts of it were already implemented in GiNaCRA [LA13], the library used for SMT-RAT. The other interval extensions are not implemented in the context of SMT-RAT but would also be suitable for our purposes.

2.1 Basic operations

In the following we define the basic arithmetic operations addition, subtraction, multiplication and division in the ICP setting. We define a real interval as follows:

Definition 2.1.1 (Interval). *An interval $I \subseteq \mathbb{R}$ is a set, such that there exists $a, b \in \mathbb{R} \cup \{-\infty, +\infty\}$ such that $I = \{x \in \mathbb{R} \mid a \sim_l x \sim_u b\}$ for some $\sim_l, \sim_u \in \{<, \leq\}$. The set of intervals is denoted by \mathbb{IR} .*

We write I as $\langle a, b \rangle$, where $\langle \in \{(\, [$ and $\rangle \in \{),]\}$. When \sim_l is $<$ we denote this by " $($ ", which means that the lower bound is not contained in the interval and call this a strict bound. When \sim_l is \leq we denote this by " $[$ ", which means that the lower bound is contained in the interval and refer to it as a weak bound. The upper bound types are defined accordingly. Note that combinations such as a lower strict and a weak upper bound are also possible.

The central idea behind interval extensions for arithmetic operations is to obtain a result interval which fulfills the following property:

No matter which number from the given interval domain you would insert into the algebraic operation without interval extension, the result is contained in the

obtained result interval

$$\forall x \in I_x, \forall y \in I_y : x \odot y \in I_x \odot I_y, \quad \odot \in \{+, -, \cdot, \div\}$$

where $I_x, I_y \in \mathbb{IR}$. Additionally, we have to consider the bound types whenever performing binary functions on intervals.

Example 2.1.1 (Boundtypes). *The intersection of two intervals*

$$[1, 3] \cap (2, 4) = (2, 3]$$

results in the maximal interval which respects both boundtypes.

With this information we are now able to explain the four basic operations needed for the ICP algorithm.

Example 2.1.2 (Interval addition). *The addition of the intervals $[2, 4]$ and $[1, 5]$ results in the interval $[3, 9] = [2 + 1, 4 + 5]$*

Example 2.1.3 (Interval subtraction). *The difference of the intervals $[2, 4]$ and $[1, 5]$ results in the interval $[-3, 3] = [2 - 5, 4 - 1]$.*

Example 2.1.4 (Interval multiplication). *The product of the intervals $[2, 4]$ and $[1, 5]$ results in the interval $[2, 20] = [2 \cdot 1, 4 \cdot 5]$*

Example 2.1.5 (Interval division). *The division of the intervals $[2, 4]$ and $[1, 5]$ results in the interval $[0.4, 4] = [2/5, 4/1]$. However, if we divide the given interval $[2, 4]$ by $[-1, 5]$, we obtain $(-\infty, -2] \cup [0.4, +\infty) = (-\infty, 2/ - 1] \cup [2/5, +\infty)$ as a result of the divisor interval containing zero.*

We encounter a problem, whenever the divisor interval contains zero. Therefore we have to introduce and apply extended interval arithmetic, which defines the division by intervals containing zero as well as definitions of the basic arithmetic functions for unbounded intervals.

According to [Kul08], the basic arithmetic operations on $A, B \in \mathbb{PR}$, where \mathbb{PR} denotes the power set (the set of all subsets) of real numbers, which \mathbb{IR} is a subset of, are defined by:

$$\bigwedge_{A, B \in \mathbb{PR}} A \circ B := \{a \circ b \mid a \in A \wedge b \in B\}, \text{ for all } \circ \in \{+, -, \cdot, /\}. \quad (2.1)$$

According to 2.1 the interval division in \mathbb{IR} is defined as

$$\bigwedge_{A, B \in \mathbb{PR}} A/B := \{a/b \mid a \in A \wedge b \in B\}. \quad (2.2)$$

Considering, that division is the inverse operation of multiplication, such that a/b is the solution of $b \cdot x = a$ we can rewrite 2.2 as

$$\bigwedge_{A, B \in \mathbb{PR}} A/B := \{x \mid bx = a \wedge a \in A \wedge b \in B\}$$

which allows to define and interpret the result of division by an interval containing zero. For more details we refer to [Kul08].

Now we can define the basic operations $+$, $-$, \cdot and $/$:

Definition 2.1.2 (Interval addition). *The addition of the intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$, where $A, B \in \mathbb{IR}$ is defined as:*

$$[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$$

Definition 2.1.3 (Interval subtraction). *The subtraction of the intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$, where $A, B \in \mathbb{IR}$ is defined as:*

$$[a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]$$

Definition 2.1.4 (Interval multiplication). *The multiplication of the intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$, where $A, B \in \mathbb{IR}$ is defined as:*

$$[a_1, a_2] \cdot [b_1, b_2] = [\min\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}, \max\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}]$$

Definition 2.1.5 (Interval division). *The division of the intervals $A = [a_1, a_2]$ and $B = [b_1, b_2]$, where $A, B \in \mathbb{IR}$ is defined as:*

$$\bigwedge_{A, B \in \mathbb{PR}} A/B := \{x \mid bx = a \wedge a \in A \wedge b \in B\}$$

where $0 \notin B$. In case $0 \in B$ we can separate eight cases as depicted in Table 2.5.

The results of an interval division with a divisor containing zero are depicted in Table 2.5. Note that the results of the division with the divisor containing zero may result in a union of two distinct intervals with a gap in between. In the ICP algorithm we handle this as a split and refer to it as a *heteronomous split* (see Section 4.5).

Note also, that the result is the smallest set containing all solutions, such that the bound types have to be adjusted accordingly.

The full behavior of the basic operations is listed in the tables 2.1 to 2.5. These tables contain the "basic" cases (bold text) as well as the exceptional cases where infinity or zero are involved. Basic means that the other cases can be derived from them by applying the rules for calculation with infinity:

$$\begin{array}{ll} \infty + x = \infty, & -\infty + x = -\infty, \\ -\infty + (-\infty) = -\infty \cdot \infty = \infty, & \infty + \infty = \infty \cdot \infty = \infty, \\ \infty \cdot c = \infty, x > 0, & \infty \cdot x = -\infty, x < 0, \\ \frac{x}{\infty} = \frac{x}{-\infty} = 0, & 0 \cdot (-\infty) = 0 \cdot (\infty) = (-\infty) \cdot 0 = (+\infty) \cdot 0 = 0 \end{array}$$

The tables represent the case that both intervals have weak lower and upper bounds. For strict bounds or combinations of weak and strict bounds the bounds of the resulting interval have to be adjusted accordingly.

Addition	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, a_2 + b_2]$	$(-\infty, a_2 + b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2]$	$(-\infty, a_2 + b_2]$	$[\mathbf{a}_1 + \mathbf{b}_1, \mathbf{a}_2 + \mathbf{b}_2]$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$(-\infty, +\infty)$	$[a_1 + b_1, +\infty)$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.1: Definition for interval addition with consideration of infinity.

Subtraction	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, +\infty)$	$(-\infty, a_2 - b_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$	$[a_1 - b_2, +\infty)$	$[\mathbf{a}_1 - \mathbf{b}_2, \mathbf{a}_2 - \mathbf{b}_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$[a_1 - b_2, +\infty)$	$[a_1 - b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.2: Definition for interval subtraction with consideration of infinity where the interval $B = [b_1, b_2]$ is subtracted from the interval $A = [a_1, a_2]$.

Division	$[b_1, b_2]$	$[b_1, b_2]$	$(-\infty, b_2]$	$[b_2, +\infty)$
$0 \notin B$	$b_2 < 0$	$b_1 > 0$	$b_2 < 0$	$b_1 > 0$
$[a_1, a_2], a_2 \leq 0$	$[\mathbf{a}_2/\mathbf{b}_1, \mathbf{a}_1/\mathbf{b}_2]$	$[\mathbf{a}_1/\mathbf{b}_1, \mathbf{a}_2/\mathbf{b}_2]$	$[0, a_1/b_2]$	$[a_1/b_1, 0]$
$[a_1, a_2], a_1 \leq 0 \leq a_2$	$[\mathbf{a}_2/\mathbf{b}_2, \mathbf{a}_1/\mathbf{b}_2]$	$[\mathbf{a}_1/\mathbf{b}_1, \mathbf{a}_2/\mathbf{b}_1]$	$[a_2/b_2, a_1/b_2]$	$[a_1/b_1, a_2/b_1]$
$[a_1, a_2], a_1 \geq 0$	$[\mathbf{a}_2/\mathbf{b}_2, \mathbf{a}_1/\mathbf{b}_1]$	$[\mathbf{a}_1/\mathbf{b}_2, \mathbf{a}_2/\mathbf{b}_1]$	$[a_2/b_2, 0]$	$[0, a_2/b_1]$
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, 0]$
$(-\infty, a_2], a_2 \geq 0$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1/b_1]$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$[0, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.3: Definition for interval division with consideration of infinity but without the divisor interval B containing zero.

Multiplication	$\frac{[b_1, b_2]}{b_2 \leq 0}$	$\frac{[b_1, b_2]}{b_1 < 0 < b_2}$	$\frac{[b_1, b_2]}{b_1 \geq 0}$	$[0, 0]$	$\frac{(-\infty, b_2]}{b_2 \leq 0}$	$\frac{(-\infty, b_2]}{b_2 \geq 0}$	$\frac{[b_1, +\infty)}{b_1 \leq 0}$	$\frac{[b_1, +\infty)}{b_1 \geq 0}$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 \leq 0$ $a_1 < 0 < a_2$	$[a_2 \cdot b_2, a_1 \cdot b_1]$ $[a_2 \cdot b_1, a_1 \cdot b_1]$	$[a_1 \cdot b_2, a_1 \cdot b_1]$ $[\min(a_1 \cdot b_2, a_2 \cdot b_1), \max(a_1 \cdot b_1, a_2 \cdot b_2)]$	$[a_1 \cdot b_2, a_2 \cdot b_1]$ $[a_1 \cdot b_2, a_2 \cdot b_2]$	$[0, 0]$ $[0, 0]$ $[0, 0]$	$[a_2 \cdot b_2, +\infty)$ $(-\infty, +\infty)$	$[a_1 \cdot b_2, +\infty)$ $(-\infty, +\infty)$	$(-\infty, a_1 \cdot b_1)$ $(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_1)$ $(-\infty, +\infty)$	$(-\infty, +\infty)$ $(-\infty, +\infty)$
$[a_1, a_2], a_1 \geq 0$ $[0, 0]$	$[a_2 \cdot b_1, a_1 \cdot b_2]$ $[0, 0]$	$[a_2 \cdot b_1, a_2 \cdot b_2]$ $[0, 0]$	$[a_1 \cdot b_1, a_2 \cdot b_2]$ $[0, 0]$	$[0, 0]$ $[0, 0]$	$(-\infty, a_1 \cdot b_2]$ $[0, 0]$	$(-\infty, a_2 \cdot b_2]$ $[0, 0]$	$[a_2 \cdot b_1, +\infty)$ $[0, 0]$	$[a_1 \cdot b_1, +\infty)$ $[0, 0]$	$(-\infty, +\infty)$ $[0, 0]$
$(-\infty, a_2], a_2 \leq 0$	$[a_2 \cdot b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_1]$	$[0, 0]$	$[a_2 \cdot b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_1]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 \geq 0$	$[a_2 \cdot b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_2]$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1 \cdot b_1]$	$(-\infty, +\infty)$	$[a_1 \cdot b_2, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1 \cdot b_2]$	$(-\infty, +\infty)$	$[a_1 \cdot b_1, +\infty)$	$[0, 0]$	$(-\infty, a_1 \cdot b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1 \cdot b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[0, 0]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.4: Definition of interval multiplication with consideration of infinity.

Division $0 \in B$	$B = [0, 0]$	$\frac{[b_1, b_2]}{b_1 < b_2 = 0}$	$\frac{[b_1, b_2]}{b_1 < 0 < b_2}$	$\frac{[b_1, b_2]}{0 = b_1 < b_2}$	$\frac{(-\infty, b_2]}{b_2 = 0}$	$\frac{(-\infty, b_2]}{b_2 > 0}$	$\frac{[b_1, +\infty)}{b_1 < 0}$	$\frac{[b_1, +\infty)}{b_1 = 0}$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 < 0$	\emptyset	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$ $\cup [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a_2/b_2]$ $\cup [0, +\infty)$	$(-\infty, 0]$ $\cup [a_2/b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 > 0$	\emptyset	$(-\infty, a_1/b_1]$ $\cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup [a_1/b_2, +\infty)$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 < 0$	\emptyset	$[a_2/b_1, +\infty)$ $\cup [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$ $\cup [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a_2/b_2]$ $\cup [0, +\infty)$	$(-\infty, 0]$ $\cup [a_2/b_1, +\infty)$	$(-\infty, 0]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 > 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 < 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 > 0$	\emptyset	$(-\infty, a_1/b_1]$ $\cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup [a_1/b_2, +\infty)$	$[a_1/b_2, +\infty)$	$(-\infty, 0]$	$(-\infty, 0]$ $\cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $\cup [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.5: Extended interval division of $A = [a_1, a_2]$ by $B = [a_1, a_2]$ with intervals containing zero and infinity.

Chapter 3

SMT solving

The goal of the presented ICP module is to speed up the search of an SMT solver for a solution for *quantifier-free nonlinear real arithmetic* (QFNRA) formulae. SAT Modulo Theories (SMT) solving for the existential fragment of nonlinear real algebra is able to cope with formulae of this type. At the beginning we present the theory fragment QFNRA and introduce definitions relevant for the remainder of this thesis. Afterwards we shortly sketch the general process of SMT-solving. In the last section we introduce the underlying SMT toolbox SMT-RAT for the presented ICP module.

3.1 QFNRA formulae

The purpose of the ICP module in the context of this thesis is to speed up the solving process of an SMT solver on *quantifier-free nonlinear real arithmetic* (QFNRA) formulae, which in the following we also refer to as formulae. QFNRA formulae are formed by the following grammar:

$$\begin{aligned} p &:= r \mid x \mid (p + p) \mid (p - p) \mid (p \cdot p) \\ c &:= p < 0 \mid p = 0 \\ \varphi &:= c \mid (\varphi \wedge \varphi) \mid \neg\varphi \end{aligned}$$

where $x \in \text{Var}(\varphi)$ denotes a variable and $\text{Var}(\varphi) = \{x_1, \dots, x_n\}$ represents the set of variables occurring in φ . Furthermore, $r \in \mathbb{Q}$ is a rational constant. Syntactic sugar, such as $p \leq 0$, $p > 0$, $p \geq 0$ and $(\varphi \vee \varphi)$, can be derived from the grammar.

We can rewrite a polynomial p created by the previously mentioned grammar as

$$p := \sum_{i=0}^n r_i \prod_{j=0}^{n_i} x_j^{e_{ij}},$$

where r_i is the rational coefficient of the monomial $\prod_{j=0}^{n_i} x_j^{e_{ij}}$. The degree of the polynomial p is defined as

$$\text{deg}(p) = \max \left\{ \sum_{j=0}^{n_i} e_{ij} \mid i \in \{0, \dots, n\} \right\}$$

The constraints can be separated into two groups: The set of linear constraints in φ , L_φ ($\text{deg}(p) \leq 1$) and the set of nonlinear constraints in φ , N_φ with $\text{deg}(p) > 1$, where p is the left-hand side of the constraint c . If N_φ is empty we call φ a quantifier-free linear real arithmetic (QFLRA) formula.

The set of variables which occur in nonlinear constraints of φ is denoted by V_{N_φ} while the set V_{L_φ} denotes variables, which occur in linear constraints. Note that the intersection $V_{N_\varphi} \cap V_{L_\varphi}$ does not have to be empty, as there might be variables which occur in both linear and nonlinear constraints.

The solving process for QFNRA formulae with current solvers is in the worst case doubly exponential. Procedures to solve QFNRA formulae are cylindrical algebraic decomposition (CAD), virtual substitution (VS) or Gröbner bases among others. The mentioned procedures have already been implemented as SMT-RAT modules. For QFLRA-solving there exist methods which are fast (polynomial) [Kha79]. The module implemented in SMT-RAT to solve QFLRA formulae is based on the Simplex algorithm [DM06, Dan98], which has an exponential worst-case complexity. However, in practice it is the fastest approach by far.

3.2 Bounded NRA

The ICP method operates on bounded intervals, which requires that all variables have to be bounded initially. In general it suffices to provide an over-approximating bound for every variable as long as this bound is given. This is required due to the behavior of the underlying interval arithmetic on unbounded intervals (see Section 2.1). Even if one bound for $x_j \in \text{Var}(\varphi)$ is given while the other bound is not, interval arithmetic operations will most likely result in completely unbounded intervals $(-\infty, +\infty)$.

Thus, we assume that each variable $x_j \in \text{Var}(\varphi)$ has a corresponding interval $[x]_{x_j} \in \mathbb{IR}$. Each interval has a lower and an upper bound such that we can separate four cases, depending on the bound types (see Chapter 2): $[x]_{x_j} = [l_j, u_j]$, (l_j, u_j) , $[l_j, u_j)$ or $(l_j, u_j]$ with $l_j, u_j \in \mathbb{R}$, $l_j \leq u_j$. We denote a point interval as $[c]$, $c \in \mathbb{R}$, which stands for $[c, c]$.

We denote a vector of intervals $[x] = ([x]_{x_1}, \dots, [x]_{x_n})^T \in \mathbb{IR}^n$ as a *search box*.

Definition 3.2.1. *A search box of a given constraint set C is a vector of intervals $[x] = ([x]_{x_1}, \dots, [x]_{x_n}) \in \mathbb{IR}^n$ for the variables occurring in C . Each interval can be represented independently by the conjunction of two linear constraints:*

$$\Phi([x]_{x_j}) = \begin{cases} l_j \leq x_j \wedge x_j \leq u_j & \text{iff } [x]_{x_j} = [l_j, u_j] \\ l_j < x_j \wedge x_j < u_j & \text{iff } [x]_{x_j} = (l_j, u_j) \\ l_j < x_j \wedge x_j \leq u_j & \text{iff } [x]_{x_j} = (l_j, u_j] \\ l_j \leq x_j \wedge x_j < u_j & \text{iff } [x]_{x_j} = [l_j, u_j) \end{cases}$$

and the search box can be represented by the conjunction of linear interval constraints:

$$\Phi([x]) = \bigwedge_{i=1}^n \Phi([x]_{x_i})$$

Without loss of generality we can assume that constraints include only the relational symbols $<$, \leq , $=$. A preprocessing in the ICP module enables us to transform all original constraints

$$c_i : \sum_{j=1}^n r_j \cdot \prod_{k=1}^{n_i} x_k^{e_{jk}} \sim 0, \quad \sim \in \{<, \leq, =\}$$

to the form

$$c'_i : h_i + \sum_{j=1}^n r_j \cdot \prod_{k=1}^{n_i} x_k^{e_{jk}} = 0 \quad (3.1)$$

such that all relations are encoded in the interval bound types of the additional variables h_i . Therefore all constraints can be transformed to the required form of equations as demanded in Section 4.4.1.

Outlook Currently the ICP module demands a bounded initial search box. It is still unresolved how to determine initial bounds efficiently. A rather inefficient way of obtaining initial bounds would be to pass all equations we gained after the preprocessing to a CAD implementation to project all polynomials occurring in the equations to univariate polynomials and then calculate Cauchy-bounds. The interval bounds would be the minimal and the maximal Cauchy-bound.

3.3 Classic SMT solving

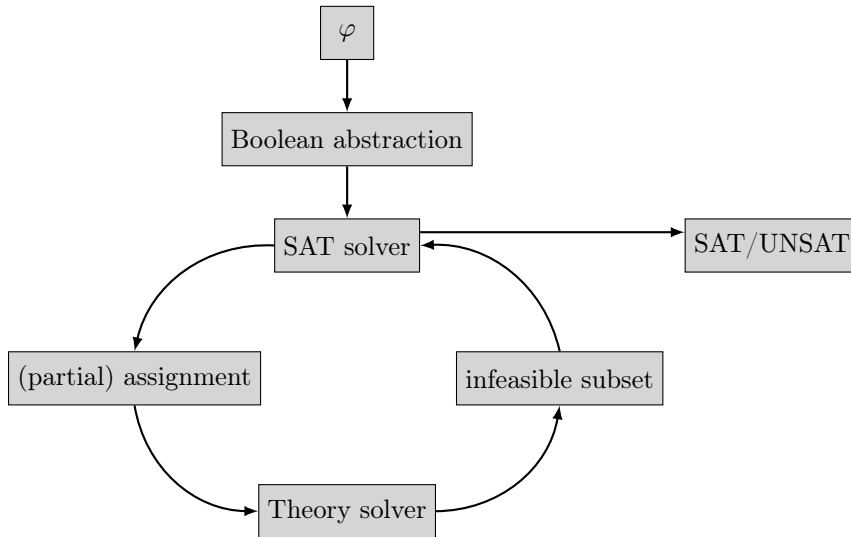


Figure 3.1: The classic SMT-solving approach.

As there are already very efficient SAT solvers, SMT-solving benefits from a combination of them, when combined with theory solving. At first a Boolean abstraction of the formula φ is created which is brought to conjunctive normal

form (CNF). The Boolean abstraction introduces a fresh Boolean variable for every constraint in the original formula and keeps the Boolean skeleton intact. Converting the formula to CNF can be done efficiently by applying Tseitin’s encoding [Tse68]. The SAT solver now tries to assign the variables of the Boolean abstraction in less-lazy DPLL style [GHN⁺04]. Each variable represents the abstraction of an original QFNRA subformula of the input formula. After each assignment the set of asserted subformulae which are constraints is checked for consistency by a theory solver. The difference to full-lazy SMT-solving is, that in full-lazy SMT-solving the SAT-solver creates a full assignment for all Boolean variables and afterwards hands the corresponding constraints over to the theory solver.

The theory solver now tries to solve the given set of constraints. If the theory solver finds a solution it informs the SAT solver about this and further decisions can be made until a complete satisfying assignment is found. If the theory solver cannot find a solution because the set of constraints is unsatisfiable it provides *infeasible subsets* of the checked set of constraints. Using this piece of information the conflict can be resolved and the search for a satisfying full assignment of the Boolean variables whose corresponding set of constraints is consistent can be continued. If no backtracking is possible as all available options have already been tried by the SAT solver the input formula is declared as unsatisfiable. Due to this behavior the underlying theory solver should support *incrementality* which means that an belated assertion or a removal of a constraints results in a minimum of adjustments and already gained information which is not affected can be used for further computation. Furthermore, if the implementation supports the creation of infeasible subsets and backtracking we refer to it as *SMT compliant*.

3.4 SMT-RAT

The context in which the ICP module we present is used is the SMT-solving toolbox SMT-RAT [CLJA12]. The toolbox provides a set of modules, among others SMT compliant implementations of Gröbner bases, the virtual substitution and the cylindrical algebraic decomposition. Furthermore, SMT-RAT maintains a manager, which can use the provided modules to combine them to a solving strategy (see Figure 3.2).

Each module maintains *received formulae* C_{rec} which is a set of QFNRA formulae the module is intended to solve. This set can be modified by using the provided functions *assert(formula φ)* and *remove(formula φ)*. The central method *isConsistent()* calls the consistency check of the actual received formulae C_{rec} . As some of the contained modules are not complete (e.g. VS and Gröbner bases) the modularity enables us to pass QFNRA formula sets to succeeding modules in case they cannot be solved by the current module. This is referred to as *calling a backend* and is performed by calling the function *run-Backends()*. When the consistency check fails it is intended to give a reason for the failure, which is determined by sets of subsets of formulae C_{inf} called the *infeasible subset*, where $C_{inf} \subseteq 2^{C_{rec}}$.

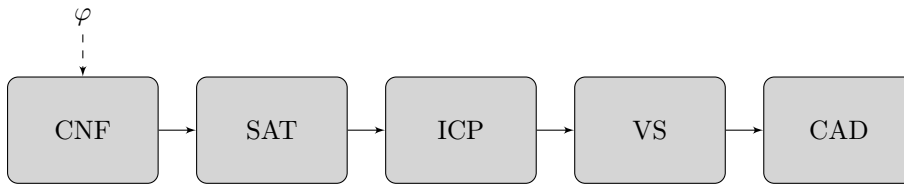


Figure 3.2: An example strategy can consist of a module which converts the input formula φ to CNF. Afterwards the SAT-solver creates a Boolean abstraction and computes a partial assignment of the variables representing the constraints. The corresponding constraints are then passed to the succeeding ICP Module. Here the bounds on the variables in the received constraints are reduced for the following modules, for instance like the VS or CAD. Note that not all modules are necessarily called. If one module finds a solution itself it does not invoke its backend module.

3.4.1 Passed and Received Formulae

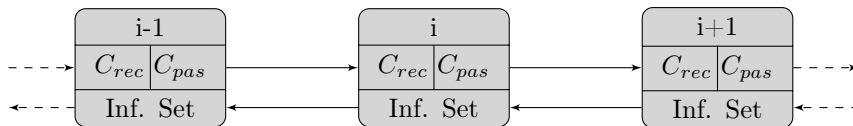


Figure 3.3: The communication of the single modules via passed (C_{pas}) and received (C_{rec}) formulae. The modules also pass their generated infeasible subset back to the calling modules.

As previously mentioned modules can pass sets of formulae to succeeding modules, which means that each module also keeps a *set of passed formulae* C_{pas} . The received formula contain the set of formulae the module has to solve. Therefore, the passed formula of the i -th module corresponds to the received formula of the $(i+1)$ -th module. Note that a set of formulae is semantically defined by the conjunction of these formulae. In the following we call a member of the received/passed formula also a *subformula* of the received/passed formula. The formulae in the received formula can be different from the formulae in the passed formula. However, if this is the case it is important to keep a mapping from subformulae of the received formula to subformulae of the passed formula. This is needed, as the returned infeasible subsets have to be subsets of the received formulae. Hence, when using the obtained infeasible subsets I_{passed} of a backend for reasoning, a module must map the formulas in I_{passed} to formulae contained in C_{rec} .

3.4.2 General module operations

Each module has to implement an interface which provides basic operations such as *assertion* and *removal* of formulae to and from the received formula as well as the consistency *check* of it. Among others, the function *inform(..)* can be called. It triggers the preprocessing and initialization of constraints inside

a module. The management of infeasible subsets also happens according to a module interface, called *getInfeasibleSubsets()*.

In the following paragraphs we sketch the purpose of the most important functions every module implements.

inform(constraint c)

The purpose of the method *inform(constraint c)* is to be able to perform all possible preprocessing before the assertion of formulae. This requires that a module is informed about all possible constraints before adding formulae with *assert(formula φ)* which contain them.

assert(formula φ)

The general difference between *assert(..)* and *inform(..)* can be described as follows: While *inform(..)* just determines the set of constraints a module will possibly encounter, *assert(formula φ)* indicates that a formula φ has been added to the received formula. During *assert(formula φ)* all operations should be performed which are needed to activate constraints of φ for a later consistency check.

remove(formula φ)

The method *remove(formula φ)* is called in order to indicate, that φ is removed from the received formulae C_{rec} . With the removal of a formula all results gained from calculations including this formula need to be removed as well. The remaining results from calculations of formulae $\psi \in C_{rec}, \psi \neq \varphi$ should stay untouched.

isConsistent()

The method *isConsistent()* is called to check the consistency of current received formulae. Note that a call to *runBackends()* conforms to calling the method *isConsistent()* of the succeeding module. The function performs the solving and returns an answer of the type "True", "False" or "Unknown". In case the answer is "False", the module should be able to provide an explanation in the form of an infeasible subset.

getInfeasibleSubsets()

The method *getInfeasibleSubsets()* is called in order to return the infeasible subsets to the preceding module in case the consistency check evaluated to *False*.

Chapter 4

The ICP module

In this section we provide background information about the Interval Constraint Propagation (ICP) module which is inspired by the approach by Gao et al. [GGI⁺10]. The module can be separated in the four major parts preprocessing, contraction, splitting and validation. First, we present the general module in the first section. The next sections cover further details of the module, starting with the preprocessing of the given constraints. The central element of the algorithm, the contraction of intervals, is described afterwards. This section also contains information about the proper selection of *contraction candidates*. When the algorithm has come to a fixpoint, an *autonomous split* can advance the procedure and provide new possibilities of progress. After a *solution candidate* has been found this candidate has to be verified during the validation part of the algorithm.

4.1 Algorithm

In this section we present the basic functionality of the ICP module. Note that the input formula of a general module is a set of subformulae, which is defined as the conjunction of the subformulae. The ICP module can only perform a consistency check on a set of constraints and returns "*Unknown*" if this is not the case. The Sections 4.2 to 4.6 deal with this consistency check, whereas Section 4.7 explains how incremental manipulation of the received formula of the ICP module is performed.

As the module is designed to be part of a whole solver strategy as well as to be usable standalone, a preprocessing has to be included. This ensures, that all constraints are put in the correct format for further processing, no matter in which way the module is used. The main idea of the preprocessing is to split nonlinear constraints from the linear ones and to transform all constraints to equations. The former is achieved by introducing a fresh real-valued variable v_m for each nonlinear monomial m , replace all occurrences of m in the considered constraint by v_m and add the equation $v_m = m$. The latter is done by introducing fresh variables to obtain equations (see Section 4.2).

After the preprocessing the actual contraction of the intervals takes place. The algorithm chooses the next possible combination of constraint and variable, we

refer to as the *contraction candidate* (see Definition 4.4.1), by a certain heuristic (see Section 4.4.2), in order to apply contraction. The algorithm repeats choosing a contraction candidate and contracting until we fulfill the *precision requirements* (see Section 4.6.2), i.e. a given target diameter for all intervals of the current search box is reached, or there is no contraction candidate contracting the search box sufficiently, what we refer to as fixpoint.

If the latter case occurs, the search box is split in half and contraction is resumed on the gained search box (see Section 4.5). After reaching a point where the precision requirements are met, the contraction is stopped and the resulting search box is handed over to the validation. After successfully validating the box against the *linear feasible region* it possibly contains a solution. We verify the existence of a solution in this box by an according backend call. In case of an unsuccessful validation either the violated linear constraints are added as contraction candidates or if already existing the box is set as (unsuccessfully) validated (see Section 4.6). If contraction leads to an empty set the box is discarded as a possible solution and the algorithm continues with the next possible box (see Section 4.5.1).

4.2 Preprocessing

Linear and nonlinear constraints are separated during the preprocessing (line 3) and ICP mainly performs on the nonlinear constraints, because firstly there are already efficient algorithms to solve linear constraints [DM06] and secondly ICP is known to suffer from the *slow convergence problem*, which can occur, when we perform ICP on linear constraints (see Example 4.4.2).

Therefore it is important to detect the nonlinear parts in the given constraint set, such that ICP initially can be applied on nonlinear constraints only. The preprocessing we implement in the ICP module is related to the approach presented by Gao et al. [GGI⁺10].

For every new nonlinear monomial m_i of the left-hand side of the considered constraint c_i (see Section 3.1) we introduce a fresh *nonlinear variable* n_i such that we obtain a new constraint $m_i - n_i = 0$. Additionally to keep the original structure, the nonlinear part in the original constraint c_i is replaced by the newly introduced variable n_i . To be able to cope with relational symbols in the now linearized constraints, we add a new slack variable s_i for every new (linearized) left-hand side, set the relation to equality and add the bound correspondingly. Assume, we have the nonlinear constraint

$$\sum_{i=0}^n c_i \prod_{j=0}^{n_i} x_j^{e_{ij}} \sim b, \sim \in \{<, \leq\},$$

we replace $\prod_{j=0}^{n_i} x_j^{e_{ij}}$ by a fresh real-valued variable m_i . For all linear constraints $\sum c_i m_i$ we equalize it by adding a slack variable s_i gaining the equation $-s_i + \sum c_i m_i$ and add the bound $s_i \sim b$. Note that we introduce only one nonlinear variable for equal nonlinear monomials and only one slack variable for equal linear left-hand sides without constant parts.

This is done because the ICP algorithm demands equations as input, which are gained by using the bounds of the slack variables to represent possible inequations.

Example 4.2.1 (Preprocessing). *Original constraints given to the module:*

$$[x^2 - 1 - y = 0 \wedge x - y \leq 0]$$

Add nonlinear variable n_1 for x^2 : First add the identity $x^2 - n_1 = 0$ and replace all occurrences of x^2 in the original constraints to obtain linearized constraints:

$$[x^2 - n_1 = 0 \wedge n_1 - 1 - y = 0 \wedge x - y \leq 0]$$

Introduce slack variables for the linear constraints:

$$[x^2 - n_1 = 0 \wedge n_1 - 1 - y = 0 \wedge x - y - s_1 = 0 \wedge s_1 \leq 0]$$

It is important to keep the mapping from the original constraints to the preprocessed ones as for the communication to the other modules we need to remember the original constraints. If we, for instance, obtain an infeasible subset from a backend and we want to use it for the construction of the ICP module’s infeasible subset, it is necessary to look up the correct preprocessed original constraints. We introduce a mapping from original constraints to linearized ones and from linearized constraints to the corresponding equations with slack variables to achieve this (see Section 4.7).

After the preprocessing, the obtained linear equations are passed to an internal LRA module implemented according to [DM06] – the method *inform(..)* of the LRA module is called with the linearized constraint (line 4). Due to the fact that the internal LRA module is implemented as an SMT-RAT module it is accessed by the same functions *inform(..)*, *assert(..)*, *remove(..)* and *isConsistent()*.

At this point we already have all information required for the creation of the nonlinear contraction candidates (line 5). Note that we use the LRA module’s internal slack variables, which are created in the aforementioned fashion. The linear contraction candidates are held in a mapping, which maps the slack variables of the LRA module to the linear contraction candidates. Furthermore, we inform the backend about the new constraints.

```

1 inform(Constraint _constr)
2 {
3   (linearConstraint, [nonlinearReplacements]) = linearize(_constr);
4   informLRASolver(linearConstraint);
5   createNonlinearCandidates([nonlinearReplacements]);
6   informBackend(_constr);
7 }

```

Listing 4.1: The method *inform(constraint)*, which is called at first to perform preprocessing.

Outlook Currently, we use the constraints from the received formula of the ICP module as the constraints for the backends. It is also possible to pass the preprocessed constraints to the backend which saves transformation time in case we need the infeasible subset returned by the backend for internal purposes and do not intend to pass the gained infeasible subset to a preceding module.

4.3 Consistency check

After the preprocessing and the assertion of subformulae (see Section 4.7 for assertion), the consistency check is called. The first step executed in the consistency check function *isConsistent()* of the ICP module is to obtain the bounds for every variable in the linearized constraints. This is done by invoking the initialization of the LRA module and afterwards reading out the bounds of the variables (line 3 of Listing 4.2). After getting the new intervals for the variables (line 5 of Listing 4.2), the consistency check of the LRA module is invoked to check linear feasibility of the linearized constraints. In case of success the actual ICP algorithm is triggered (line 12 of Listing 4.2). Otherwise, the gained infeasible subset is passed to the previous module (line 10 of Listing 4.2).

When the ICP module finds a search box fulfilling the precision requirements, this box is validated (line 13 of Listing 4.2). In case the validation fails, a new box is selected (line 15 of Listing 4.2), otherwise the box is handed over to the backend module, which means that the consistency check of this backend is called with the current search box together with the received formula (line 20 of Listing 4.2). The same backend call is invoked, if the validation rejects a search box due to numerical errors (see Section 4.6). Depending on the solution returned of the backend, either a new box is selected for contraction (in case the backend rejects the passed search box, line 24 of Listing 4.2) or the ICP module returns *"True"* (in case the search box is accepted by the backend, line 29 of Listing 4.2). Note that if the backend returns an infeasible subset, which does not contain any bound of the current search box, it can directly be passed to the preceding module (line 27 of Listing 4.2). At this point we know that the check of the current constraint set in combination with the current box did not fail because of the passed box but because of the constraint set itself. Otherwise if a constraint defining the passed box is contained in the infeasible subset, a new box is selected and the old box is rejected.

```

1 Answer isConsistent() {
2     initializeLRA(); // creates intervals for variables
3     variables = getVariablesFromLRA();
4     for (var in variables) {
5         currentBox[var] = getIntervals(var);
6     }
7     answer = isConsistentLRA();
8     if (answer == false) {
9         infeasibleSubset = getInfeasibleSubsetFromLRA();
10        return answer;
11    }
12    solutionCandidateBox = contract(currentBox)
13    answer = validateSolution(solutionCandidateBox);
14    if (answer == false) {
15        currentBox = selectNextBox();
16        if (currentBox == NULL)
17            return false;
18    } else {
19        pushBoundsToPassedFormula(solutionCandidateBox);
20        answer = callBackends();
21        if (answer == false) {
22            createInfeasibleSubset();
23            if (infeasibleSubsetContains(solutionCandidateBox))
24                currentBox = selectNextBox();
25            else
26                infeasibleSubset = getInfeasibleSubsetFromBackend();

```

```

27     return false;
28   }else{
29     return answer;
30   }
31 }
32 }

```

Listing 4.2: The procedure *isConsistent* calls the ICP algorithm and manages also the validation as well as the calling of backends.

The detailed methods used during the consistency check are described during the following sections starting with the contraction of intervals which is performed by the ICP algorithm giving this module its name.

4.4 Contraction/ICP

In our version of the ICP algorithm, the contraction of intervals is performed by an interval extension of Newton's method [Moo77, HR97]. In general each contraction takes a constraint as well as a variable occurring in the constraint and uses this to contract the interval of the variable. We refer to this combination as a *contraction candidate*:

Definition 4.4.1. A contraction candidate c is a tuple

$$c = \langle f_i, x_j \rangle, f_i = 0 \in N_\varphi \cup L_\varphi, x_j \in \text{Var}(f_i) \quad (4.1)$$

where $\text{Var}(f_i)$ denotes the set of variables contained in f_i . Note that the constraint $f_i = 0$ is taken from the set of preprocessed constraints $N_\varphi \cup L_\varphi$ which are generated from the input formula φ .

Each contraction candidate holds a weight which measures the importance of this candidate during the past contractions and is updated after each contraction according to a weighting function (see Section 4.4.2). Having chosen a contraction candidate, the actual contraction of the interval of the corresponding variable can be started.

```

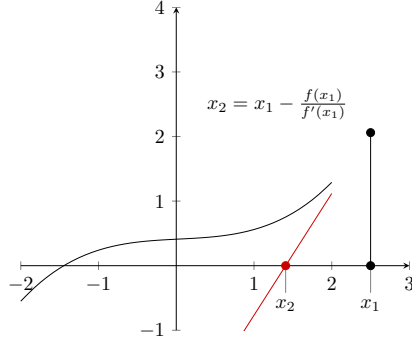
1 contract (IntervalBox _box) {
2   relevantCandidates = updateRelevantCandidates ();
3   while ( !relevantCandidatesEmpty () && !targetSizeReached ) {
4     candidate = chooseCandidate (relevantCandidates);
5     splitOccured = Newton (candidate, _box);
6     if (!splitOccured)
7     {
8       addContractionToHistoryNode (candidate);
9       addAllAffectedCandidatesToRelevant ();
10    }
11    else
12    {
13      performSplit (candidate);
14    }
15    updateWeight (candidate);
16    relevantCandidates = updateRelevantCandidates ();
17  }
18 }

```

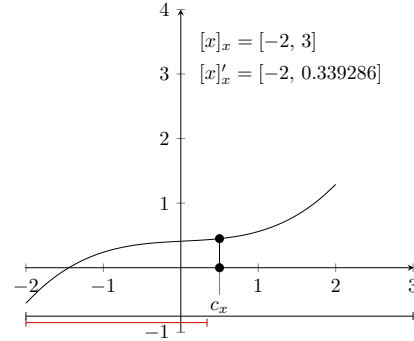
Listing 4.3: Contraction in pseudo-code. The function *updateRelevantCandidates* keeps the sorted mapping of relevant candidates up to date (see Paragraph 4.4.2 for details).

4.4.1 Newton's method

[univariate Newton][Newton's method for univariate polynomials.]



[interval Newton][The interval extension of Newton's method. Note that $[x]'_x = N_{cmp}([x]_x, f(x), x) \cap [x]_x$. The colored



markers represent the intervals.]

The idea of a contraction operator based on Newton's method can be derived as follows (the idea is based on the approach initially presented by Moore [Moo77] and further enhancements were made by Hansen and Pullman [Han78] among others):

The set of constraints from the input formula can be seen as an n -dimensional system of multivariate polynomial equations of the form

$$f(x) = (f_1(x), \dots, f_n(x))^T = 0, f : D \rightarrow \mathbb{R}^n \text{ and } f_i(x) : D \rightarrow \mathbb{R}, D \subseteq \mathbb{R}^n \quad (4.2)$$

which is ensured by the preceding preprocessing. Newton's original method is defined on univariate real-valued polynomial functions. Thus by assigning a value to $n - 1$ of the n variables in each function f_i we obtain a function of the required format and can apply Newton's method.

Similar to Newton's original method, the Newton operator for interval-valued functions can be derived from the mean-value theorem. The derivation of the Newton operator was presented by Herbot and Ratz [HR97]. As stated, we get a one-dimensional function for each f_i by fixing all but one variable x_j

$$\tilde{f}_{ij}(x_j) := f_i(x_1^*, \dots, x_{j-1}^*, x_j, x_{j+1}^*, \dots, x_n^*) \quad (4.3)$$

where $x_k^*, k \in \{1, \dots, n\}, k \neq j$ may vary in their corresponding interval $[x]_{x_k} \in \mathbb{IR}$. Applying the mean-value theorem, we get

$$\exists \xi \in [x]_{x_j} : \tilde{f}_{ij}(c_j) - \tilde{f}_{ij}(x_j) = f'_{ij}(\xi) \cdot (c_j - x_j) \quad (4.4)$$

where c_j denotes the center of the interval $[x]_{x_j}$ and it holds that $x_j \in [x]_{x_j}$. Furthermore, f'_{ij} denotes the partial derivative $\frac{\partial f_i}{\partial x_j}$. Assuming that $x^* \in \mathbb{R}^n$ is a root of f and that $\frac{\partial f_i}{\partial x_j} \neq 0$, we can transform Equation 4.4 to

$$x_j^* = c_j - \frac{\tilde{f}_{ij}(c_j)}{\tilde{f}'_{ij}(\xi)} \quad (4.5)$$

because x^* is a root of f_i as well. If we replace the indeterminate ξ by the whole interval $[x]_{x_j}$ we end at

$$\mathcal{N} := c_j - \frac{\tilde{f}_{ij}(c_j)}{\tilde{f}'_{ij}([x]_{x_j})}, \quad (4.6)$$

$$x_j^* \in \mathcal{N}$$

As stated above, the partial derivative should not be equal to zero. However, in this setting it can happen that the resulting interval of the derivative contains zero. In this case we make use of extended interval division (see Table 2.5) which results in an interval with a gap (a *heteronomous split*, see Section 4.5). Nevertheless, we do not have any information of the remaining zeroes contained in x^* – the only information we have is that they are contained in their intervals $[x]_{x_j}$, $j \in \{1, \dots, n\}$ which defines the search box $[x] = ([x]_{x_1}, \dots, [x]_{x_n})^T \in \mathbb{IR}^n$. If we replace each x_i^* , $i \in \{1, \dots, n\}$ by its corresponding interval, due to inclusion monotonicity we obtain a superset of \mathcal{N} :

$$\mathcal{N} \subseteq c_j - \frac{f_i([x]_{x_1}, \dots, [x]_{x_{j-1}}, c_j, [x]_{x_{j+1}}, \dots, [x]_{x_n})}{\frac{\partial f_i}{\partial x_j}([x]_{x_1}, \dots, [x]_{x_n})} \quad (4.7)$$

Note that the subtraction of an interval from a number (here: c_j) can be handled by treating c_j as a point interval such that $c_j - [a, b] = [c_j, c_j] - [a, b] = [c_j - b, c_j - a]$.

We are now able to define the interval Newton operator, which uses the i -th equation of the equation system to treat the j -th component of the *search box*:

Definition 4.4.2. *Let $D \subseteq \mathbb{R}^n$, $f : D \rightarrow \mathbb{R}^n$, $f = (f_1, \dots, f_n)^T$ be a continuously differentiable function, and let $[x] = ([x]_{x_1}, \dots, [x]_{x_n})^T \in \mathbb{IR}^n$ be an interval vector with $[x] \subseteq D$ and $i, j \in \{1, \dots, n\}$. Then the component-wise interval Newton operator N_{cmp} is defined by:*

$$N_{cmp}([x], i, j) := c_j - \frac{f_i([x]_{x_1}, \dots, [x]_{x_{j-1}}, c_j, [x]_{x_{j+1}}, \dots, [x]_{x_n})}{\frac{\partial f_i}{\partial x_j}([x]_{x_1}, \dots, [x]_{x_n})} \quad (4.8)$$

The arguments of the component-wise interval Newton operator are the reason for the structure of the contraction candidates which contain the needed parameters i and j (see Definition 4.4.1). Herbot and Ratz elaborated two important properties concerning the existence of a zero of the component-wise interval Newton operator N_{cmp} [HR97]:

Theorem 4.4.1. *Let $D \subset \mathbb{R}^n$, $f : D \rightarrow \mathbb{R}^n$ be a continuously differentiable function, and let $[x] = ([x]_{x_1}, \dots, [x]_{x_n})^T \in \mathbb{IR}^n$ be an interval vector with $[x] \subseteq D$. Then the component-wise interval Newton operator N_{cmp} has the following properties:*

1. Let $x^* \in [x]$ be a zero of f , then we have for arbitrary $i, j \in \{1, \dots, n\}$:
 $x^* \in ([x]_{x_1}, \dots, [x]_{x_{j-1}}, N_{cmp}([x], i, j), [x]_{x_{j+1}}, \dots, [x]_{x_n})$
2. If $N_{cmp}([x], i, j) \cap [x]_{x_j} = \emptyset$ for any $i, j \in \{1, \dots, n\}$ then there exists no zero of f in $[x]$.

Proof. Herbot and Ratz prove their first statement by referring to the derivation of the N_{cmp} operator. Let $x^* \in [x]$ be a zero of f , and let $i, j \in \{1, \dots, n\}$. If we take f_i as a one-dimensional real-valued function in x_j , we can derive from the mean-value theorem (see Equation 4.4):

$$x_j^* = c_j - \frac{f_i(x_1^*, \dots, x_{j-1}^*, c_j, x_{j+1}^*, \dots, x_n^*)}{\frac{\partial f_i}{\partial x_j}(x_1^*, \dots, x_{j-1}^*, \xi, x_{j+1}^*, \dots, x_n^*)}, \quad \xi \in [x]_{x_j} \quad (4.9)$$

with $c_j := m([x]_{x_j})$ and assuming that $\frac{\partial f_i}{\partial x_j}(\dots) \neq 0$. As we know that $x^* \in [x]$ and especially $x_j^* \in [x]_{x_j}$ we do not lose validity if we replace the unknown ξ by the whole interval $[x]_{x_j}$ – in fact we include x_j^* . If we additionally replace every x_k^* , $k \neq j$ by its corresponding interval $[x]_{x_k}$ (as we do not know the exact component x_k^* but the same statement $x_k^* \in [x]_{x_k}$ holds) we get a superset of the previous inclusion:

$$x_j^* \in c_j - \frac{f_i(x_1^*, \dots, x_{j-1}^*, c_j, x_{j+1}^*, \dots, x_n^*)}{\frac{\partial f_i}{\partial x_j}(x_1^*, \dots, x_{j-1}^*, [x]_{x_j}, x_{j+1}^*, \dots, x_n^*)} \quad (\text{inclusion}) \quad (4.10)$$

$$\subseteq c_j - \frac{f_i([x]_{x_1}, \dots, [x]_{x_{j-1}}, c_j, [x]_{x_{j+1}}, \dots, [x]_{x_n})}{\frac{\partial f_i}{\partial x_j}([x]_{x_1}, \dots, [x]_{x_{j-1}}, [x]_{x_j}, [x]_{x_{j+1}}, \dots, [x]_{x_n})} \quad (4.11)$$

$$= N_{cmp}([x], i, j) . \quad (4.12)$$

As only the j -th component of $[x]$ is treated we can conclude that

$$x^* \in ([x]_{x_1}, \dots, [x]_{x_{j-1}}, N_{cmp}([x], i, j), [x]_{x_{j+1}}, \dots, [x]_{x_n}) . \quad (4.13)$$

By using the first statement of the theorem the second one can be proven using contradiction. We assume that there exists a zero $x^* \in [x]$ of f and we assume that $N_{cmp}([x], i, j) \cap [x]_{x_j} = \emptyset$. If we apply the Newton operator with some arbitrary $i, j \in \{1, \dots, n\}$ we get:

$$x_j^* \in N_{cmp}([x], i, j) \subseteq N_{cmp}([x], i, j) \cap [x]_{x_j} = \emptyset . \quad (4.14)$$

This neglects the assumed zero and proves the second statement by contradiction. \square

Both statements from Theorem 4.4.1 have an important meaning for the application of the component-wise interval Newton operator in the ICP algorithm for contraction. The first statement ensures that regardless of how often an interval is contracted by a contraction candidate, we do not lose a solution – all zeros in the initial *search box* are always contained in the resulting search box. The second statement is useful in rejecting possible search boxes: If the consecutive application of the component-wise interval Newton operator on a given system of equations results in an empty set, the original search box did not contain any zeros. This has two important implications for the data structure of

the ICP module: Firstly, if we are able to contract a box to the empty interval we can drop the whole box and, secondly, it suffices to store the contractions which are applied on a certain box instead of storing each resulting box after one contraction step. This is because if we are able to create the empty set via contraction we can drop all previously made contractions until the point where the box has been created, e.g. by a split (see Section 4.5.1 for details of the data structure).

After contraction, the result is intersected with the original interval of the variable before contraction. Therefore, the resulting interval is at least as wide as the original one. This is done to prevent the method from diverging. Due to this approach, if the actual setup tends to diverge, the resulting relative contraction equals zero and thus, the contraction candidate is rated down. The rating of contraction candidates is done via the relative contraction and the weight is updated after each successful contraction (see Section 4.4.2). Successful in this case means that the relative contraction is above a predefined threshold and the contraction did not result in an empty interval.

Example 4.4.1 (Contraction). *We choose the constraint set from previous examples but omit the preprocessing to increase readability: $C = [c_1 : x^2 - y = 0 \wedge c_2 : x - y = 0]$. The initial search box is set as: $[x] \in \mathbb{IR}^2 = [1, 3]_x \times [1, 2]_y$. If we consider the two contraction candidates $\langle c_1, x \rangle$ and $\langle c_2, y \rangle$ and alter their application, we gain a contraction sequence. We show the first contraction in detail with the previously mentioned Newton operator:*

$$\begin{aligned} N_{cmp}([1, 3]_x \times [1, 2]_y, x^2 - y = 0, [1, 3]_x) &= [2, 2] - \frac{[2, 2]^2 - [1, 2]}{2 \cdot [1, 3]} \\ &= [2, 2] - \frac{[2, 3]}{[2, 6]} \\ &= [0.5, 1.66667] \end{aligned}$$

what results in an updated interval for x : $[1, 3] \xrightarrow{c_1^x} [1, 3] \cap [0.5, 1.66667] = [1, 1.66667]$. If we now alter the contraction candidates we obtain the sequence:

$$\begin{aligned} [1, 3]_x &\xrightarrow{c_1^x} [1, 1.66667]_x \xrightarrow{c_1^x} [1, 1.3]_x \xrightarrow{c_1^x} [1, 1.14135]_x \xrightarrow{c_1^x} \dots \xrightarrow{c_1^x} [1, 1]_x \\ [1, 2]_y &\xrightarrow{c_2^y} [1, 1.66667]_y \xrightarrow{c_2^y} [1, 1.3]_y \xrightarrow{c_2^y} [1, 1.14135]_y \xrightarrow{c_2^y} \dots \xrightarrow{c_2^y} [1, 1]_y \end{aligned}$$

As mentioned in Section 4.2, the presented preprocessing approach separates nonlinear and linear constraints. This is done due to the fact that ICP is vulnerable to *slow convergence*. We can outline this behavior by an example:

Example 4.4.2 (Slow convergence). *Consider the constraint set $C = [c_1 : y = x + 1 \wedge c_2 : x = y + 1]$. If we limit the initial box, e.g.: $[x] \in \mathbb{IR}^2 = [1, n]_x \times [1, n]_y$, $n \in \mathbb{N}$, we observe, that the constraint system is unsatisfiable. However the contraction sequence of the ICP algorithm would look like*

$$\begin{aligned} [1, n]_x &\xrightarrow{c_2^x} [2, n]_x \xrightarrow{c_2^x} [3, n]_x \xrightarrow{c_2^x} [4, n]_x \xrightarrow{c_2^x} \dots \\ [1, n]_y &\xrightarrow{c_1^y} [2, n]_y \xrightarrow{c_1^y} [3, n]_y \xrightarrow{c_1^y} [4, n]_y \xrightarrow{c_1^y} \dots \end{aligned}$$

until the system is finally declared unsatisfiable. A linear solver would find this fact without taking $2n$ contraction steps.

Outlook The contraction as described above is the first approach to contract intervals. Herbort and Ratz [HR97] mention extensions to the presented component-wise Newton operator: The usage of index lists is one of them.

The introduction of index lists aims at improving the choice of the best possible contraction candidate. The idea is to use automatic differentiation to determine all partial derivatives, which corresponds to a single evaluation of the Jacobian matrix of the system. This is done because if the derivative equals zero (the denominator in Equation 4.8) and the numerator contains zero the result will be $(-\infty, +\infty)$, which does not contribute to the solution and, thus, can directly be avoided.

According to [HR97] it is sufficient to choose only candidates from the obtained Jacobian matrix with entries different from zero. This results in a set of possible contraction candidates. However, the order of those candidates is still to be determined. The idea is to successively use all possible variables on one constraint as this results in an interval at least as small as if we choose the optimal variable directly. In fact it can be even smaller – the next optimal step can be in the set of remaining candidates

Remark 4.4.1 (Index Lists). *Let $\mathcal{S} := (\langle c_i, x_1 \rangle, \dots, \langle c_i, x_n \rangle), j \in \{1, \dots, n\}, n := |\text{Var}(c_i)|$ be the unknown optimal contraction sequence for the constraint c_i where each variable $x_k, k \in \{1, \dots, n\}$ occurs exactly once. It can be assured that the resulting relative contraction c when using a sequence $\mathcal{S}' \neq \mathcal{S}$ is at least as big as if the first contraction candidate of \mathcal{S} is chosen directly.*

Herbort and Ratz propose the usage of two index lists L_1 and L_2 . The first one contains all pairs (i, j) , where the entry in the calculated Jacobian is different from zero while the second list contains all pairs where extended interval division has to be applied.

The items for L_1 are picked starting from the diagonal element of the calculated Jacobian matrix and proceeding downwards and jumping to the first row if the bottom is reached.

The list L_2 is created by choosing the element with the largest diameter from each column whose entry is equal to zero.

4.4.2 Choice of the Contraction Candidate

During the contraction of the given search box we can choose between several contraction candidates, as we usually have more than one asserted constraint with more than one variable. However, not all candidates will result in the same relative contraction. Even worse, the possible relative contraction is dependent on the order and appearance of the previous contractions.

This means that, on the one hand, we cannot predict the possible relative contraction of a contraction candidate and, on the other hand, the possible relative contraction changes during time. Goualard and Jermann [GJ08] have related this problem to a standard problem in reinforced learning, the *multi-armed bandit problem*. This problem is described as follows.

There are k slot machines with an unknown probability to win and a fixed time horizon. The goal is to find a sequence of levers to pull such that the outcome is maximized in the given time horizon. In our case the slot machines are the contraction candidates and the outcome is the relative contraction. There

exists a non-stationary version of the problem where the probabilities of the slot machines to win change non-deterministically while time passes. The non-stationary version of this problem is closely related to our problem as the resulting contraction is not predictable until the actual calculation is done and varies during time.

If we consider our equation system of n equations with at most n variables, we have to choose between n^2 contraction candidates before every single contraction. As previously mentioned, there is no way to predict the relative contraction of a candidate such that we stick to heuristics. Goualard and Jermann [GJ08] propose a reinforced learning approach for this problem which we adapt.

The basic idea of this learning approach is to rate every applied contraction such that the weight $W^{(ij)}$ of every contraction candidate represents its importance during the past solving process. Goualard proposes an update formula whose result represents the average of the last contractions biased with a factor α which introduces a parameter to adjust the effect of the last applied contraction on the whole weight

$$W_{k+1}^{(ij)} = W_k^{(ij)} + \alpha(r_{k+1}^{(ij)} - W_k^{(ij)}) \quad (4.15)$$

where $r_{k+1}^{(ij)}$ represents the last contraction and $W_k^{(ij)}$ the previous weight of the candidate. The factor α has a strong influence on how the weights evolve: Is α close to 0, for example $\alpha = 0.1$, the initial weight has a longterm influence on the weights. On the other hand, if we pick a value close to 1, e.g. $\alpha = 0.9$, the weights change with faster pace and the last payoffs have a stronger impact on the weighting. In our approach we choose a value of $\alpha = 0.9$, as the importance of certain candidates in our examples varied a lot. Nevertheless, this factor is one of the many relevant parameters which can be tuned to change the behavior of the whole algorithm. Therefore, the optimal setting of this parameter still has to be determined.

After picking a contraction candidate, the actual contraction is performed with the component-wise interval Newton operator N_{cmp} (see Section 4.4.1). After the contraction, the relative contraction is computed and the weight of the chosen contraction candidate is updated.

ICP-relevant candidates Currently we keep a sorted mapping of weights to contraction candidates, which we refer to as "ICP-relevant candidates". In this mapping it is easy to pick the candidate with the highest weight, as it is located at the last position in the mapping. Due to the improbable but still possible case, that two candidates have the same weight, we extended the key of the ICP-relevant candidates to a tuple $(weight, id)$, where the id is set upon creation of the contraction candidate by an internal manager.

If a relative contraction above a certain threshold has been made, the candidate is kept in the mapping, otherwise it is removed. Additionally, all candidates which contain the variable whose interval has changed and are active (see Section 4.6), are added to the ICP-relevant candidates.

This ensures that all candidates which could profit from a changed interval are enlisted and are at least considered once for contraction. This implies, that each

change is eventually propagated throughout the whole system, as every affected contraction candidate, which is active is added as a *relevant candidate*.

Outlook As previously mentioned, the influence of the parameter α may be of interest during the proper selection of the next contraction candidate. Furthermore, Herbort and Ratz introduced index lists which also might be of interest to improve the algorithm [HR97]. In their paper Goualard and Jermann propose to use priority queues for every variable which should ensure that each variable is eventually used for contraction [GJ08]. The current approach via the weights allows, that a limited number of contraction candidates are chosen alternately, which might suppress the reduction of all variables. However, the current implementation also stops reducing one variable if the target diameter of its interval has been reached such that this suppression is only temporary. Another small optimization is to use linear contraction candidates only once as the relative contraction of a linear candidate equals zero if it is applied more than once consecutively.

4.5 Split

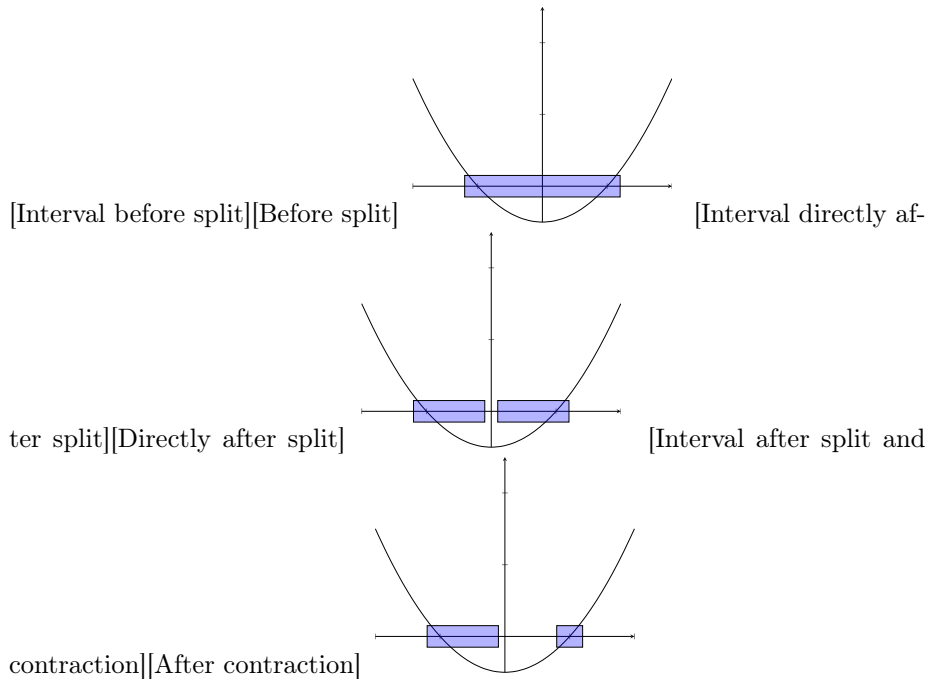


Figure 4.1: The split enables us to perform additional contractions without losing a solution. Note that the gap in the second figure is just for graphical representation.

In case the algorithm has reached a fixpoint during the contractions, we perform a split (see Figure 4.1). Currently, we define that a fixpoint in the algorithm is reached when the ICP-relevant candidates mapping is empty. This

means that the last possible contraction has been less than a certain threshold. Therefore, the corresponding contraction candidate has been removed from the mapping.

At this point no useful contraction can be applied. However, the search box might still be too large to suffice our *precision requirements* (see Section 4.6.2), which only happens, if we stop contraction due to loss of progress. To be able to continue, we split the actual search box in the middle of an interval domain, which is larger than our precision requirement, in one randomly chosen dimension. This splitting can be seen as a contraction, such that all candidates which contract in this dimension update their weights with a payoff of 50% (see Section 4.4.2).

The second case when a splitting can happen is, when the contraction results in a *heteronomous split* (see Sections 2.1.5 and 4.4). In this case, the payoff is bigger or equal to 50% due to a possible gap in the resulting intervals, which reduces the interval even more. Nevertheless, the treatment is the same as if an *autonomous split* had happened, which includes creating two new search boxes and enlisting them into the *history tree*.

4.5.1 History Tree

To keep track of splits (*heteronomous* and *autonomous*), we introduce a tree-based structure to store solver states. It is necessary to keep track of solver states due to the previously mentioned splittings. Every split produces a pair of new boxes. Therefore, it is crucial to distinguish each box from the other ones, as each box can be treated separately. The actual solver state is represented by the current search box. Initially, the module starts with one root node containing the initial search box and a right child, which is set as the first considered node. This is done to be able to recall the initial box.

All contractions are applied on the current node and the applied contractions are stored in the node. Whenever a split happens, two nodes are created and appended to the current node. Each new node contains one half or less of the original search box. Additionally, the old node sets a variable indicating in which dimension it has been split.

To be able to get infeasible subsets, all nodes keep track of their applied contractions, such that we can apply a simple backtracking mechanism if needed. This includes that a set of references to the used contraction candidates is stored, where each candidate occurs at most once.

When selecting a new state in the tree, the current considered intervals concerned for contraction are set to this state (see also lines 15 and 24 in Algorithm 4.2). The selection of the next state follows a left-most depth-first traversal of the tree. When selecting a new box, the older nodes left of the new node can be deleted. This keeps the tree in an almost linear structure, as every split results in two nodes from which only one is considered. Furthermore, the old nodes are not of interest any more as they were already visited.

Outlook Currently, the splitting creates an almost linear binary tree. Autonomous splits divide an interval into two equal parts by cutting in the middle of the interval of the desired dimension.

One variation of this approach is to split in more than two parts. This would

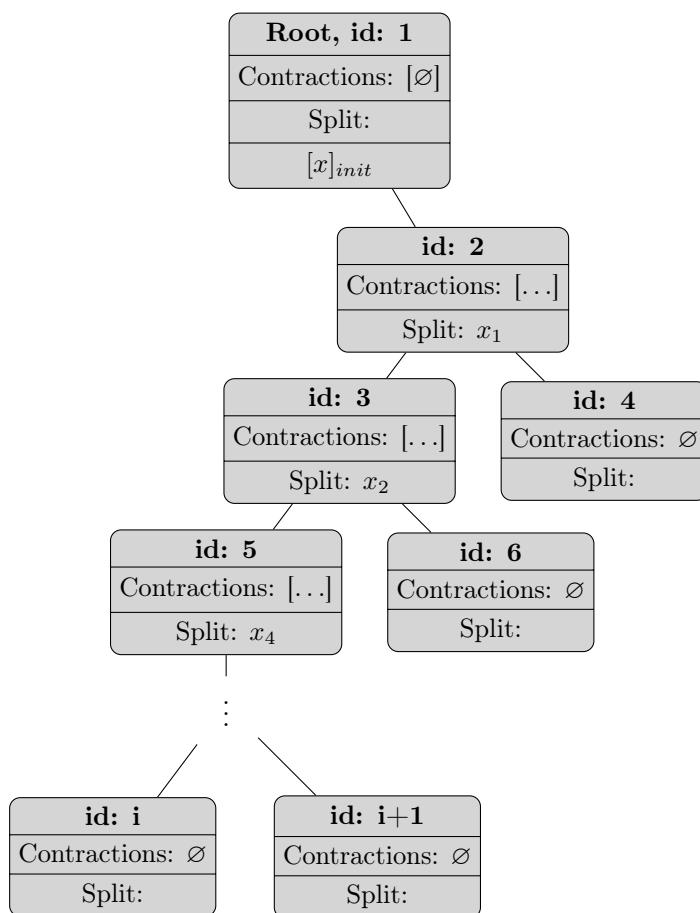


Figure 4.2: The HistoryTree holds all important information to switch and restore a solver state. Each state keeps track of the applied contractions and in case a split happens, the variable where the split occurred is saved. Note that we omitted the search boxes each node keeps.

result in a tree structure with a higher branching rate and the relative contraction by an autonomous split would be increased (e.g. splitting into three equal parts results in 66% relative contraction).

Another possible improvement is to raise the selection of the next box after a split to a SAT solver. Whenever a splitting decision has been made, it is possible to create deductions of the form

$$\begin{aligned}
 & [(\bar{A}_{i-1} \vee \bar{B}_{i-1} \vee \neg x < r \vee B_i) \vee \\
 & (\bar{A}_{i-1} \vee \bar{B}_{i-1} \vee \neg x \geq r \vee B'_i)] \wedge \\
 & \quad (x' < r \vee x' \geq r) \wedge \\
 & \quad (\neg x' < r \vee \neg x' \geq r),
 \end{aligned}$$

where A_i denotes the set of active constraints, B_i denotes the future search box and r is the proposed split. This deduction indicates, that from the actual

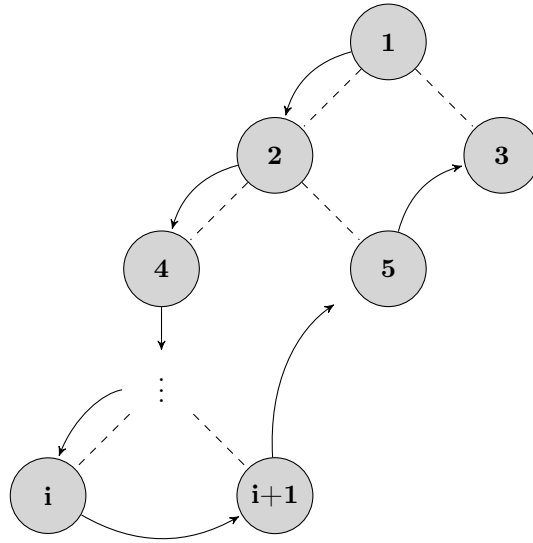


Figure 4.3: The choice of the next solver state is done in a left-most depth-first manner. Note that when switching the solver state all visited nodes are cut to keep the overall number of nodes small.

active constraints A_{i-1} and the actual search box B_{i-1} a split in dimension x is proposed at point r , which either results in the search box B_i or B'_i , but not both at the same time. This tautology is passed to a SAT solver, which might use this and other information to select the next box with more information than the ICP module has at this point.

4.6 Validation

After obtaining a search box which suffices our needs in terms of size, it is necessary to validate this box against the *linear feasible region*, represented by the conjunction of all linear constraints. We have to consider three cases which can occur:

1. The *search box* resides completely inside the *linear feasible region*
2. The *search box* resides partially inside the *linear feasible region*
3. The *search box* lies completely outside the *linear feasible region*

It is especially important to separate the first case from the second. To do so, Gao et. al. have introduced a two-phase validation which we use as well [GGI⁺10].

In the first step, to separate the third case from the other two cases, we consider an arbitrary point of the resulting box and check whether it is contained in the *linear feasible region*.

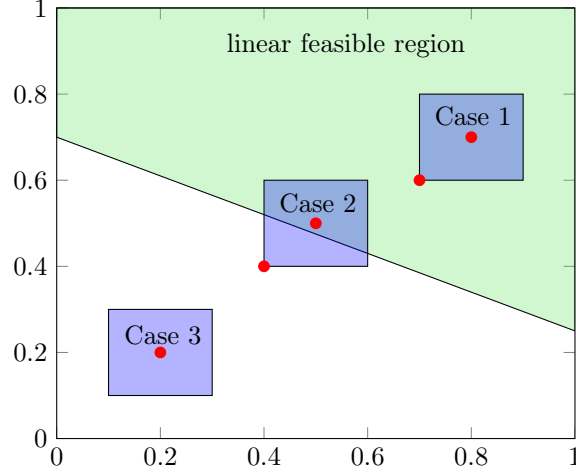


Figure 4.4: The three cases we want to separate during validation: Case 1, where the center and the rest of the box lies inside the linear feasible region, Case 2 which is distinguished from case 3 by the maximal point and Case 3 which we distinguish using the center point.

Definition 4.6.1 (Linear feasible region). *The linear feasible region determines the solution space resulting from the conjunction of the linear constraints*

$$\bigwedge_{i=1}^n \left(\sum_{j=1}^k a_{ij} y_j = s_i \wedge s_i \sim e_i \right), \sim \in \{<, \leq, \geq, >\}$$

where e_i is the constant part and $V_{L\varphi} := \{y_1, \dots, y_k\}$ are the k variables occurring in the linear constraint with their coefficients a_{ij} .

If this check fails, we either have a case-two scenario where the considered point occasionally lies outside the linear feasible region, or the box is completely outside such that any chosen point inside the box would violate at least one linear constraint. As it does not matter which point is chosen, we simply take the center of the search box as the point which is checked in the first phase (line 3 of Listing 4.4). The checking of a point can be performed by the internal LRA module. To this end, all linearized constraints as well as constraints representing the center point of the search box are handed over to the LRA module (lines 4 and 5 of Listing 4.4)

$$\bigwedge_{x_i \in V_{N\varphi}} \left(\frac{u_i - l_i}{2} = 0 \right) \wedge L_\varphi \wedge \bigwedge \Lambda,$$

where the set $V_{N\varphi} := \{x_1, \dots, x_n\}$ denotes the set of variables, which occur in nonlinear constraints of φ , Λ denotes the set of already asserted constraints and l_i, u_i are the lower respectively upper bound of $[x]_{x_i}$. In case the center point of the search box lies inside the linear feasible region, the LRA module returns a point-solution \vec{y} for all variables $\{y_1, \dots, y_m\} \in V_{L\varphi} \setminus V_{N\varphi}$, which only occur in linear constraints (line 6 of Listing 4.4).

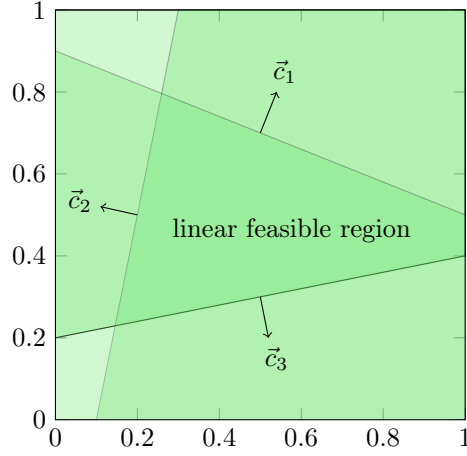


Figure 4.5: The linear feasible region as a conjunction of linear constraints $(c_1 \wedge c_2 \wedge c_3)$, each defining a half-space.

The second phase of the validation process separates the second case from the first one. The goal is to verify characteristic points of the search box against the linear constraints. As the linear feasible region is determined by the intersection of the linear constraints, it suffices to check each linear constraint separately. The point solution \vec{y} , obtained from the LRA module, is used to set all remaining variables, which only occur in the linear constraints L_φ (line 13 of Listing 4.4). We can rewrite $L_\varphi \wedge \bigwedge \Lambda$ as the intersection of half-spaces where we separate the linear and nonlinear variables:

$$L_\varphi \wedge \bigwedge \Lambda \equiv \bigwedge_{j=1}^k \vec{c}_j^T \vec{x} \leq e_j + \vec{d}_j^T \vec{y} \quad (4.16)$$

The vector \vec{c}_j contains all coefficients of the nonlinear variables, such that $\vec{c}_j = (c_{j1}, \dots, c_{jn})$ and the vector $\vec{d}_j = (d_{j1}, \dots, d_{jm})$ contains the coefficients of the linear variables.

If the conjunction is satisfied, this means that the chosen point for the variable x_j lies in the *linear feasible region*. To check the actual search box, it is sufficient to validate only the maximal points of the box (lines 16 and 17 of Listing 4.4).

Lemma 4.6.1 (Maximal points). *Given c_j and $[x] := [l_1, u_1] \times \dots \times [l_n, u_n]$ the maximal points of $[x]$ are*

$$\max \left\{ \vec{c}_j^T \vec{p} \mid \vec{p} = (p_1, \dots, p_n) \in [x] \text{ and } p_i = \begin{cases} l_i & \text{if } c_{ji} \leq 0 \\ u_i & \text{if } c_{ji} > 0 \end{cases} \right\}.$$

The intuitive idea behind Lemma 4.6.1 is, to pick the point of the box farthest in the direction of the linear constraint and verify it against this constraint. If this point lies outside the half-space depicted by the linear constraint, we can be sure that the actual search box covers the linear feasible region only partially. Note that we only perform the second check if the center point of the search box

is verified. Thus, at least the center point lies inside the linear feasible region, while the maximal point lies outside - we encounter a case two scenario.

```

1 Validate(searchBox, assertedLinearConstraints)
2 {
3   centerConstraints = centerPoint(searchBox);
4   assertLRA(centerConstraints);
5   assertLRA(assertedLinearConstraints);
6   pointSolution = isConsistentLRA();
7   if ( isEmpty(pointSolution) ) // the centerpoint lies outside
8   {
9     violatedConstraints = getInfeasibleSubsetLRA();
10  }
11  else // validate maximal points
12  {
13    linearVariables = pointSolution;
14    for(constraint in assertedLinearConstraints) do
15    {
16      p = MaximalPoint(searchBox, assertedLinearConstraints);
17      answer = validatePoint(p);
18      if (answer == False )
19      {
20        addToViolatedConstraints(constraint);
21      }
22    }
23  }
24  return violatedConstraints;
25 }

```

Listing 4.4: Validation algorithm in pseudo-code.

During the validation there are two steps where the linear constraints might be violated: Either, while validating the center point (line 6 of Listing 4.4) or during the validation of the maximal points (line 17 of Listing 4.4). In each case, the validation algorithm returns a set of linear constraints. In order to consider the linear feasible region during the further solving process, we add the violated linear constraints to the constraints relevant for contraction, if they are not already contained. The contraction candidates of those constraints are set as active to track whether they were already considered during validation. Note that firstly, all nonlinear contraction candidates are always active and secondly, the state active is different from "being asserted" (asserted linear contraction candidates are only considered for contraction if they have been activated before during validation).

In a future contraction, the added active linear constraints are considered as well. If they are again violated during a later validation phase, they can be neglected as this violation is only because of numerical errors. This can be assumed, because if the constraints are already contained in the *relevant candidates* set, they are at least once considered for contraction before the next validation is called. Note that validation is only called if the set of relevant candidates is empty and thus, every contraction candidate inside it has been unsuccessfully used for contraction at least once.

If a linear constraint, which is already activated (and thus has been in the relevant candidates set) is violated due to the validation process, the violation can be neglected, as it can only occur due to numerical errors, as ICP produces a solution box which fulfills all considered constraints (see Theorem 4.4.1).

4.6.1 Involving backends

Whenever a search box has been validated, it is a *solution candidate* for the succeeding backends. As the idea of the ICP module as a part of a solver strategy is to reduce the search space for the subsequent backends, the latter are called with solution boxes which usually are smaller than the initial problem. To call a backend with a solution box, the constraints representing the box are added to the *passed formulae*.

If the backends declare the passed formulae as inconsistent, they provide an infeasible subset. The infeasible subset is a subset of the constraints contained in the passed formulae. This means that it can also contain constraints representing the solution box. If this is not the case, the infeasible subset obtained by the backend can be re-transformed to the constraints from the *received formulae* of the ICP module and can be used as the infeasible subset.

However, if the infeasible subset of a backend contains constraints, which are part of the constraints representing the solution box, it is necessary to select a new box, as this box has been invalidated.

If we have already tried all possible boxes the ICP module returns "*False*" and uses the whole received formula as an infeasible subset.

Outlook Another approach, which has to be implemented yet, may result in generally smaller infeasible subsets. If we use the information gained from the infeasible subsets of child nodes to create an infeasible subset for the parent node in the history tree, we could optimize the process of infeasible subset generation and might exclude branches in the tree early.

4.6.2 Precision

Precision is one of the tunable parameters in the ICP module. We define a threshold for the contraction, which we refer to as target diameter. It determines the maximal size of the solution candidate, which is handed over to the backend. However, this parameter is crucial for the interaction of the ICP module and its backends. On the one hand, the backend profits from a smaller size of the received solution candidate box. For example, the CAD module might be able to reduce the set of considered polynomials or the VS module can drop certain substitution candidates. This results in a faster solving in the backend.

On the other hand, the calculation of a smaller solution candidate box requires more contraction and splitting steps. We assume, that this marks a trade-off, as there might be a point where the gain of the contractions is less effective than the reduction by the backend. However, this is still to be evaluated and furthermore, this point might differ from constraint system to constraint system.

4.7 Incrementality

So far, we only considered a fixed set of constraints. As previously mentioned we apply less-lazy SMT-solving, which implies that the ICP module should support incrementality. This means, that the set of constraints can change. In SMT-RAT this is realized via the functions *assert(formula φ)* and *remove(formula φ)*.

4.7.1 assert(formula φ)

When the assertion of a constraint is called, the linearized constraint is asserted in the internal LRA module (line 4). Furthermore, the original constraint is asserted in the backend (line 5). Note that assert has a parameter of type formula, which contains an original constraint. This requires, that during the preprocessing a mapping from original constraints (referred to as origins) to modified constraints has to be kept. This is necessary in order to identify the created nonlinear contraction candidates in case, that the original constraint contains nonlinear parts (line 6).

Example 4.7.1 (Origins). *During inform(..) the constraints*

$$(x^2 + x - y = 0 \wedge x^3 + 2x^2 - y = 0)$$

have been passed to the module and the nonlinear contraction candidates $\langle h_{r_0} - x^3 = 0, x \rangle, \langle h_{r_0} - x^3 = 0, h_{r_0} \rangle, \langle h_{r_1} - x^2 = 0, x \rangle, \langle h_{r_1} - x^2 = 0, h_{r_1} \rangle$ have been created. In case the function `assert($x^2 + x - y = 0$)` is called, the nonlinear contraction candidates $\langle h_{r_1} - x^2 = 0, x \rangle, \langle h_{r_1} - x^2 = 0, h_{r_1} \rangle$ have to be identified by their corresponding origins and are activated.

These nonlinear contraction candidates are activated during this step (line 7). Furthermore, the linear contraction candidates are created during assertion if they do not already exist. At this point, the LRA module should be informed about all possible constraints and thus has created the according slack variables (line 8). These are collected by the ICP module and assigned to the newly created linear contraction candidates (line 9). The intervals of the slack variables taken from the LRA module are the intervals for the slack variables of the linearized constraints. However, the intervals for slack variables are not calculated yet, as the assertion of further constraints can have influence.

Note that it can occur, that one contraction candidate is representing more than one constraint. This happens especially for the nonlinear contraction candidates. Consider the set of asserted constraints:

$$(x^2 + x - y = 0 \wedge x^3 + 2x^2 - y = 0)$$

During `inform(..)` the nonlinear replacements $h_{r_0} - x^3 = 0$ and $h_{r_1} - x^2 = 0$ with their corresponding contraction candidates $\langle h_{r_0} - x^3 = 0, x \rangle, \langle h_{r_0} - x^3 = 0, h_{r_0} \rangle, \langle h_{r_1} - x^2 = 0, x \rangle, \langle h_{r_1} - x^2 = 0, h_{r_1} \rangle$ are created. The contraction candidates which replace x^2 are now asserted twice. This is handled by a counter which is increased during assertion if the candidate already exists and decreased upon removal of an original constraint.

```

1  assert(Formula _subformula)
2  {
3    linearFormula = getLinearizedConstraints(_subformula);
4    assertLRASolver(linearFormula);
5    assertBackend(_subformula);
6    nonlinearCandidates = findNonlinearCandidates(_subformula);
7    activateCandidates(nonlinearCandidates); //increases counter if active
8    slackVariable = getLRASlackvariables(_subformula);
9    createContractionCandidate(_subformula, slackVariable);
10 }
```

Listing 4.5: The assertion procedure in pseudo-code.

4.7.2 `remove(formula φ)`

During the solving phase, the preceding module can remove formulae, for example during backtracking and indicate, that after the call of `remove(..)` the constraint will be removed from the received formulae and thus is irrelevant for the next call of `isConsistent()`.

When the function `remove(formula φ)` is called, we require the mapping from original to preprocessed constraints to identify the corresponding contraction candidates (linear and nonlinear ones). The counter of a contraction candidate is decreased in case it is > 1 . Otherwise, the contraction candidate is set to inactive and thus will not be considered for contraction until it is asserted again (see Section 4.4).

4.7.3 Data Structure

The needed data structure to realize incrementality includes several aspects. First of all, contraction candidates are separated into linear and nonlinear contraction candidates. Furthermore, the contraction candidates can be asserted, such that we have to keep track of the asserted ones, as they are the only ones which should be considered for contraction. Each contraction candidate can be active or inactive. This property is important whenever contraction candidates are considered for contraction. Nonlinear candidates are always active, while linear ones only become active after validation in case their corresponding linear constraints were violated (see Section 4.6 for details).

In case new constraints are asserted or removed, we need a mapping from original constraints to preprocessed constraints. This is needed, because the constraints contained in the subformulae, which are passed with `assert(..)` or `remove(..)` are original constraints and we need to update the corresponding contraction candidates, which have been created from the preprocessed constraints.

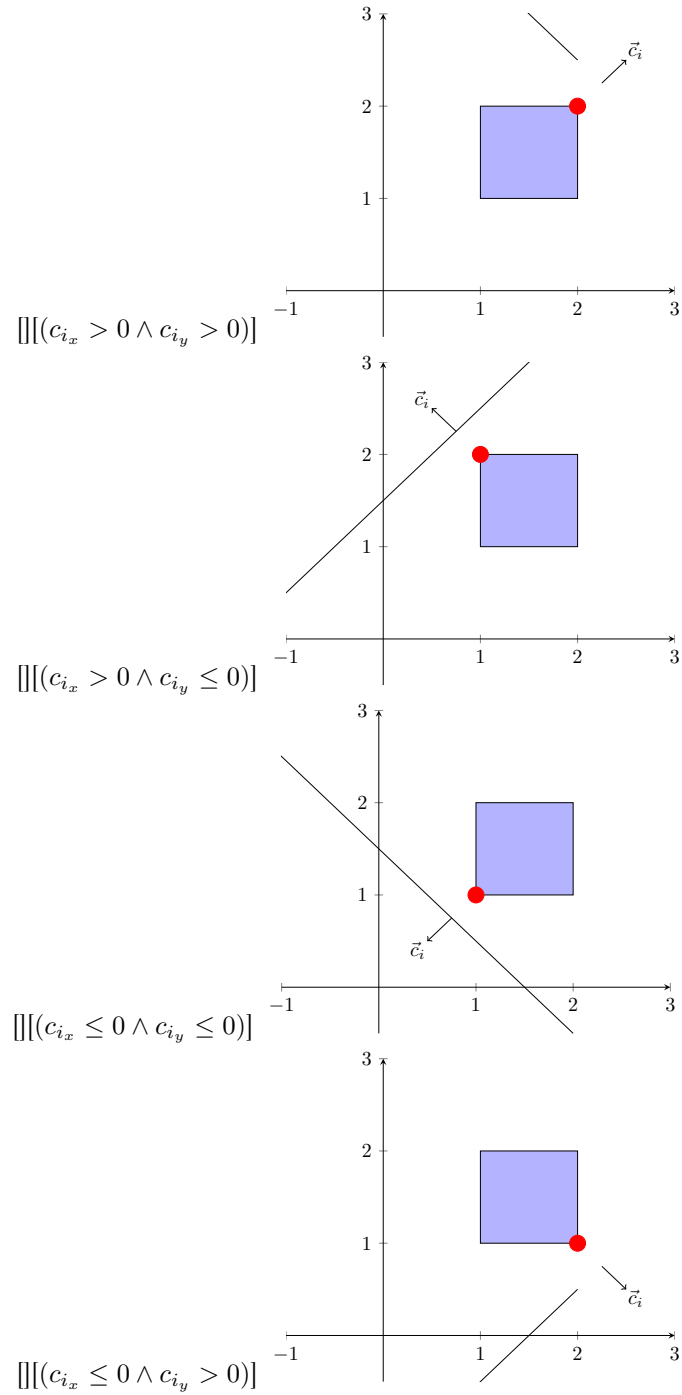


Figure 4.6: The maximal points in relation to different constraints.

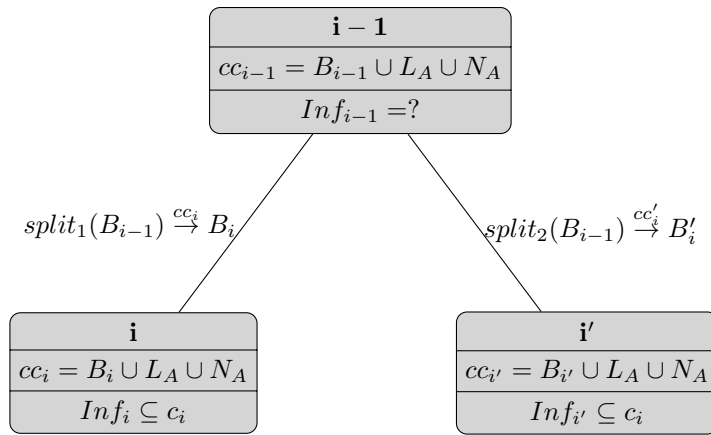


Figure 4.7: The challenge for finding the infeasible subset of a falsified parent node.

Chapter 5

Conclusion

In this chapter we show an exemplary run of the designed ICP module with different target diameters for the solution candidate boxes and discuss the results. Furthermore, we sum up the ideas for future work as also presented in the last chapters. A final summary concludes this thesis.

5.1 Results

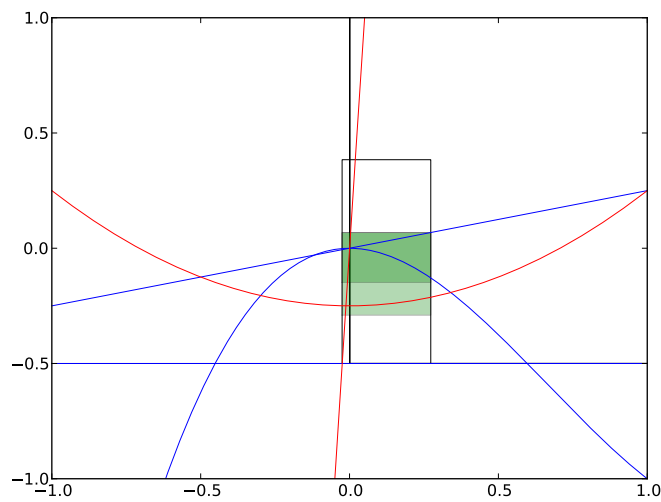


Figure 5.1: An extract of an exemplary run of Example 5.1.1 with an initial box $[x] = [-1000, 1000]_x \times [-1000, 1000]_y$ and a target diameter of 1. The resulting boxes are drawn without filling, the dropped ones are colored red and the boxes which are passed over to the backend are colored green. The intersection of the green boxes shows, that in fact two boxes have been passed to the backend.

Example 5.1.1. For first testing we created a simple input formula which covers most interesting cases:

The required constraint system should contain linear as well as nonlinear constraints. Furthermore, incrementality should be tested with a disjunction, where at least one removal and one assertion are possible.

$$\begin{aligned} & \left[\left(x^3 - 2x^2 - y > 0 \right) \right. \\ & \wedge \left(\frac{1}{2} + y \geq 0 \right) \\ & \wedge \left(\frac{1}{4}x - y \geq 0 \right) \\ & \wedge \left(-4x^2 + 20x - y = 0 \vee \frac{1}{2}x^2 - 1 - y = 0 \right) \\ & \left. \wedge \left(x^2 - y = 0 \vee \frac{1}{2}x^2 - \frac{1}{4} - y = 0 \right) \right] \end{aligned}$$

The test was done with an initial box $[x] = [-1000, 1000]_x \times [-1000, 1000]_y$ which added the constraints for the boundaries

$$\begin{aligned} & \left[(x \leq 1000) \right. \\ & \wedge (x \geq -1000) \\ & \wedge (y \leq 1000) \\ & \left. \wedge (y \geq -1000) \right] \end{aligned}$$

to the constraint set.

We tried different target diameters for the solution candidate boxes starting from 10 decreasing by factor 10 until we reached the target diameter 0.0001. The described test run was made by the strategy previously presented in Section 3.4 which consists of the modules

$$\varphi \rightarrow \text{CNF} \rightarrow \text{SAT} \rightarrow \text{ICP} \rightarrow \text{VS} \rightarrow \text{CAD}$$

The modules VS and CAD are succeeding the ICP module and thus operate as supporting backends. The results of the test runs are depicted in the following table

\emptyset	backend calls	inv. boxes	depth avg(max)	# contr.	# splits
10	8	1	28(64)	155	100
1	7	2	25(66)	219	120
0.1	4	5	36(82)	369	177
0.01	7	9	39(94)	525	218
0.001	4	17	43(106)	638	247
0.0001	4	22	48(122)	735	281

Table 5.1: The results of the test run show, that with decreased target diameter the number of contractions and splits increases, while the number of backend calls only changes marginally.

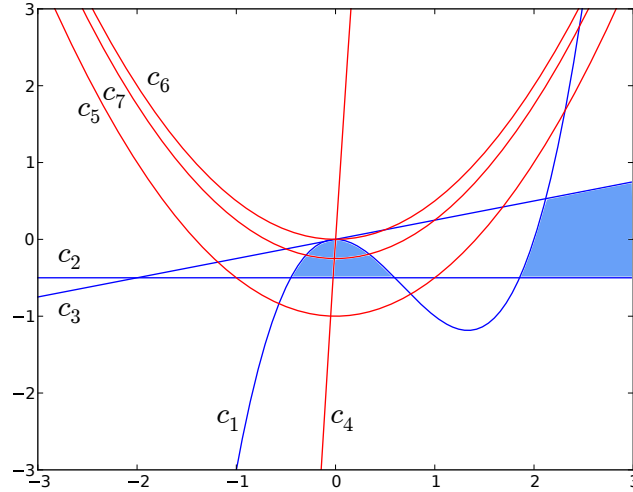


Figure 5.2: The solution space is denoted by the blue area, which is gained by an intersection of half-spaces created by two linear and one cubic inequality (blue). The actual solution is the intersection between the quadratic functions (red), which lies inside the solution space.

The area covered by boxes, which are handed over to the backend is visualized in Figure 5.1 by the green boxes with a target diameter of 1. In this case there did not occur any rejections of boxes in the visible extract.

The solution candidate boxes, which should be generated to a diameter of 1 are in this case smaller than 1. This can be explained with the fact, that the contraction is done stepwise while the contraction per step cannot be predicted as already mentioned (see Section 4.4.2). Therefore, if a box is marginally larger than the target diameter, it often occurs that the next contraction results in an interval smaller than the target diameter.

The results for the presented example with different target diameters are depicted in Table 5.1. Note that with a smaller target diameter the number of splits and contractions rises as expected. The average depth of the tree also satisfies our expectations. The system has, due to the preprocessing at most 7 variables, which all require a search box of the demanded target diameter (2 original variables (x, y), 2 variables for nonlinear replacements and 5 variables as linear slack variables). Without contraction, the tree depth is limited by the number of splits until the target diameter is reached. We can calculate the upper limit for the tree depth when using binary splits as $\sum_{x \in Var(\varphi)} \log_2([x]_x^{init})$, where $[x]_x^{init}$ denotes the initial interval for x .

The number of backend calls does not change significantly. This can be explained by the fact, that a smaller target diameter but the same set of constraints just requires more contractions and splits per step, than the same set with a larger target diameter. Note that a smaller box diameter speeds up the backend calls.

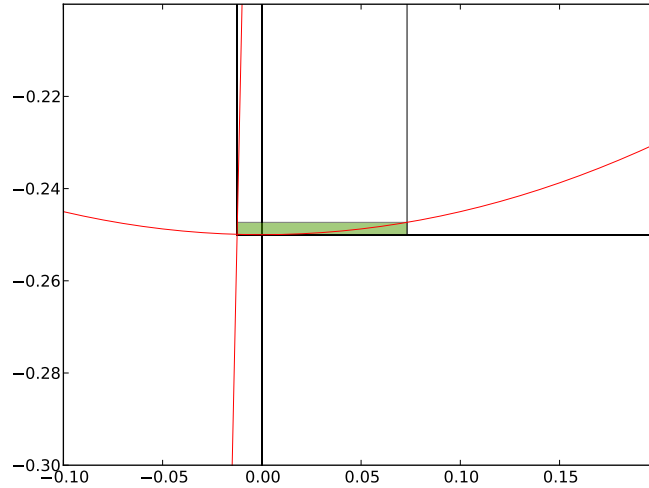


Figure 5.3: Another extract of a run with the example constraint set, made with a target diameter of 0.1. The diameter of the box in the y-dimension is by far smaller than the target diameter, which results of the unpredictable outcome of the contractions.

5.2 Future work

The presented ICP module has lots of options for optimizations. The current state of the module provides the basic behavior of a module in the context of SMT-RAT [CA11]. However, most of the operations performed can be optimized and improved.

- **Contraction and choice of contraction candidates:** The choice of contraction candidates currently depends on the weighting. An optimization is to improve the weighting itself, meaning, that the influences, which affect the weight are extended (e.g. number of variables, number of occurrences in the input formula, degree of the polynomial). Also the initial weight can be adjusted according to preliminary information concerning the constraints themselves. This is closely related to the usage of index lists encouraged by Herbot and Ratz [HR97] (see Section 4.4f).
- **Splitting:** The splitting also offers options for optimization. A mentioned idea was, to experiment with more than binary splits to increase the relative contraction gained by the split and to flatten the resulting history tree. It is also mentioned, to raise the splitting decision to the SAT solver by creating tautologies, such that the decision for the chosen box after the split can be done by the SAT solver (see Section 4.5).
- **History tree:** In the current version, the history tree is reset whenever a change to the constraint set happens. However, in case a constraint is added we could use the already existing information of the search boxes

and further decrease diameter by cutting it with the results from the contractions according to the new constraint set (see Section 4.5.1).

- **Infeasible subset generation:** The current approach does not include any intelligence concerning the infeasible subsets. Therefore it offers a lot of improvements as on the one hand minimized infeasible subsets might decrease the number of theory calls, as the SAT solver has more information. On the other hand, by creating more sophisticated infeasible subsets, we also might deduct reductions of the maintained history tree and thus drop boxes earlier (see Section 4.6.1).

5.3 Summary

The goal of this thesis was to develop a module for the SMT-RAT toolbox, which uses ICP as a mechanism to efficiently reduce search boxes for given QFNRA formulas.

We presented an SMT-RAT module, which combines ICP and an LRA solver module to achieve the mentioned goal. In this context, we introduced general background knowledge about SMT-RAT and the SMT solving procedure. We presented the existential fragment of nonlinear real arithmetic, which is the underlying logic for the input formulas. Furthermore, we outlined the needed information about interval arithmetic, which is crucial to perform the ICP algorithm.

We showed the general structure of the designed module as well as technical details concerning the single parts it contains. We sketched the preprocessing of the input formula to separate linear and nonlinear constraints. We showed how the interval contraction works in the ICP algorithm and provided a proof of its validity. The splitting as a mechanism, which helps to reduce intervals where simple contraction does not show progress, was shown after the contraction. In combination with the splitting, the history tree was introduced to store the solver states resulting from the splits. We showed a lightweight validation procedure to validate possible solution candidate boxes before passing them to the backend.

We gave ideas for optimizations for most of the parts of the presented module. At last we used a small example to present the correct functioning of the module and to visualize the progress during the decision process of the created module. We thereby also sketched the influence of the target diameter of the solution candidate boxes on the behavior of the module.

Bibliography

- [BKW93] Thomas Becker, Heinz Kredel, and Volker Weispfenning. *Gröbner bases: A computational approach to commutative algebra*. Springer, 1993.
- [CA11] Florian Corzilius and Erika Ábrahám. Virtual substitution for SMT-solving. In *Proceedings of the 18th International Conference on Fundamentals of Computation Theory (FCT'11)*, volume 6914 of *Lecture Notes in Computer Science*, pages 360–371. Springer, 2011.
- [CLJA12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox. In *Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *Lecture Notes in Computer Science*, pages 442–448. Springer, 2012.
- [Col74] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *SIGSAM Bull.*, 8(3):80–90, August 1974.
- [Dan98] George B. Dantzig. *Linear Programming and Extensions*. Princeton Landmarks in Mathematics and Physics Series. Princeton University Press, 1998.
- [DM06] Bruno Dutertre and Leonardo De Moura. Integrating Simplex with DPLL(T). Technical report, CSL, SRI INTERNATIONAL, 2006.
- [FHT⁺07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.
- [GB06] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, March 2006.
- [GGI⁺10] Sicun Gao, Malay Ganai, Franjo Ivančić, Aarti Gupta, Sriram Sankaranarayanan, and Edmund M. Clarke. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'10)*, 2010.

- [GHN⁺04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.
- [GJ08] Frédéric Goualard and Christophe Jermann. A reinforcement learning approach to interval constraint propagation. Technical report, University of Nantes, 2008.
- [Han78] Eldon R. Hansen. Interval forms of Newton's method. *Computing*, Volume 20, Issue 2:153–163, 1978.
- [HR97] Stefan Herbort and Dietmar Ratz. Improving the efficiency of a nonlinear-system-solver using a componentwise Newton method. Technical report, Universität Karlsruhe, 1997.
- [Kha79] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.
- [Kul08] Ulrich W. Kulisch. Complete interval arithmetic and its implementation on the computer. In *Numerical Validation in Current Hardware Architectures*, volume 5492 of *Lecture Notes in Computer Science*, pages 7–26. Springer, 2008.
- [LA13] Ulrich Loup and Erika Ábrahám. Ginacra: Ginac real algebra package website (<http://ginacra.sourceforge.net/>), March 2013.
- [Moo77] Ramon E. Moore. A test for existence of solutions to non-linear systems. *SIAM Journal on Numerical Analysis*, 14:611–615, 1977.
- [Tse68] Grigori S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.
- [VHMK97] Pascal Van Hentenryck, David McAllester, and Deepak Kapur. Solving polynomial systems using a branch and prune approach. *SIAM J. Numer. Anal.*, 34(2):797–827, April 1997.
- [Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5:3–27, 1988.