Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF COMPUTER SCIENCE

USING FOURIER-MOTZKIN VARIABLE ELIMINATION FOR MCSAT EXPLANATIONS IN SMT-RAT

Lorena Calvo Bartolomé

Prüfer:
Prof. Dr. Erika Ábrahám
Apl. Prof. Thomas Noll

Zusätzlicher Berater:
M.Sc. Gereon Kremer

Aachen, 14.09.2018

Abstract

Satisfiability Modulo Theories (SMT) constitutes one of the most outstanding approaches in the field of *satisfiability checking*. *SMT problems over the real numbers* are of special concern because of their importance in verification, theorem proving and design of hybrid systems. The usual methods of solving linear arithmetic are either based on *Fourier-Motzkin elimination* or the *Simplex algorithm*. The *model-constructing satisfiability calculus (MCSAT)* is a novel development in the context of SMT solving. Whereas the CAD method is already implemented as explanation generation in the MCSAT approach for non-linear arithmetic, a MCSAT approach for linear real arithmetic is not available. Since the Simplex method is not well-suited for explanation generation, a quantifier elimination method is needed.

Although the *Fourier-Motzkin variable elimination* method can be quite expensive in practice, the aim of this paper is presenting a *MCSAT approach for linear real arithmetic* using the Fourier Motzkin variable elimination as explanation generation. This implementation is integrated as a theory solver into the MCSAT framework of the SMT solver *SMT-RAT*. As conclusion, the efficiency of the approach is analyzed.

Dedicated to my grandmother,
for whom I am still at the school.

Acknowledgements

First of all, many thanks to Gereon Kremer for the continuous help with this thesis, and to the Prof. Dr. Erika Ábrahám for trusting me from the first moment and giving me the opportunity to write my thesis about this interesting topic. Furthermore, I would like to express my gratitude towards Prof. Thomas Noll for agreeing to be my second supervisor. Finally, I wish to express my deep gratitude to my family. Thanks to my parents for always supporting and encouraging me, and always guiding me in the right direction. Last but not least, thanks to Patrick for never leaving my side and putting up with my bad mood in the last months.

Contents

1 Introduction **9**
1.1 Thesis overview 10

2 Preliminaries **11**
2.1 Technical background 11
2.2 Linear real arithmetic 11

3 Satisfiability checking **14**
3.1 Satisfiability (SAT) solvers 14
3.2 Satisfiability-modulo-theories (SMT) solvers 16
3.3 Quantifier Elimination Methods 17
3.4 A model-constructing satisfiability calculus 18
3.5 SMT-RAT 22

4 The Fourier-Motzkin Elimination **23**
4.1 Algorithm description 25

5 Using Fourier-Motzkin Elimination for explanations in MCSAT **29**
5.1 Elimination of bounded variables 31
5.2 Intervals generation and conflict determination 32
5.3 Selection of the "best" conflict 33
5.4 Explanation generation 33

6 Experimental results **37**
6.1 General performance 37
6.2 Running time performance on individual instances 39
6.3 Final analysis 40

7 Conclusion **41**
7.1 Summary 41
7.2 Future work 41

Chapter 1

Introduction

Over the last decades, the increasingly technological development is exerting profoundly changes on the way people live and work. It is impacting all disciplines, economies and industries, and consciously or unconsciously, technical systems leverage almost every aspect of our everyday lives. We are talking about breakthroughs in key areas such as artificial intelligence, robotics, self-driving cars, autonomously acting missiles, 3D printing, nanotechnology, biotechnology... All of the last examples show us until which level our society trusts in technology. However, both complexity and potential risk of the specified tasks are being increased to such an extent that ensuring an accurate performance of the software is critical. The latest is done in terms of *software verification*, that is, the process of checking whether a software system meets specifications and fulfills its intended purpose.

During the process of *software verification*, the program is commonly transformed into a mathematical model, and the properties to be proven are formalized on the basis of this model. That is how we come to the term of *satisfiability checking*, which aims to develop algorithms and tools for checking the satisfiability of logical formulas. In the late '90s, an impressive progress in this area was made in terms of propositional logic, resulting in powerful *SAT solvers*. Driven by this success, propositional SAT solving starting to be enriched with solver modules for different theories, coming up with *SMT solvers*.

The common way to solve a *SMT problem* is employing a *SAT solver* to *enumerate* the assignments of the *Boolean abstraction* of the input formula. Then a *decision procedure*, dedicated to reasoning about conjunctions of theory-specific constraints, is used to either confirm or refute the candidate Boolean assignment. This extension of the *DPLL algorithm* to incorporate reasoning about a theory T is called *DPLL(T)*.

In the last few years, the idea of *direct model construction* complemented with *conflict resolution* has been successfully generalized to fragments of SMT dealing with theories including linear real/integer arithmetic, nonlinear arithmetic and floating-point. Although all these procedures are quite effective in their corresponding first-order domains, they are not completely accepted since their limitations in purely reasoning and incompatibility with DPLL(T). The *model-constructing satisfiability calculus* or *MCSAT* [DMJ13] is a recent development in SMT solving, which includes all the previous mentioned decision procedures while resolving the previous mentioned limitations.

In 2012, the research group Theory of Hybrid Systems at the RWTH Aachen, lead by Prof. Dr. Erika Ábrahám, started the development of *SMT-RAT* [CLJÁ12], its own SMT solving engine. SMT-RAT is built on C++, and its main goals are extensibility, modularity and flexibility; moreover, it is provided as free software. At the beginning, the main focus of SMT-RAT was solving problems based on theories considering non-linear real and integer arithmetics. Nonetheless, the project has as ambition extending its capabilities towards other theories. Currently, there are two strategies of SMT-RAT, namely *SMT-RAT* and *SMT-RAT-MCSAT*.

The *simplex methods* can be used to check sets of *linear real arithmetic* constraints for satisfiability, whereas the *cylindrical algebraic decomposition (CAD)* can be used for *non-linear real arithmetic*. There is also another procedure that could be considered for linear real arithmetic, namely *Fourier-Motzkin variable elimination*.

There already exists an MCSAT approach for non-linear arithmetic using the cylindrical algebraic decomposition for explanation generator, which was implemented in Z3, Yices2 and SMT-RAT, but there is no MCSAT approach available for linear real arithmetic. The simplex method has as disadvantage that it is not well-suited for explanation, since for this task we need a *quantifier elimination method*.

1.1 Thesis overview

In this thesis, an adaptation of the Fourier-Motzkin elimination as theory solver for explanation generator within the SMT-RAT MCSAT framework was implemented.

This work begins with the presentation of some technical background information in **Chapter 2** necessary for a better understanding of the thesis. Also, the basics of LRA are presented. In **Chapter 3** we give a brief explanation about satisfiability solving, and we go through the concepts of SAT and SMT solving. After that, we present the quantifier elimination methods and give a brief introduction about the MCSAT approach. We finish the chapter by particularizing on the SMT-RAT solver. The basic principles of the Fourier-Motzkin elimination are described in **Chapter 4**. It is followed in **Chapter 5** by the description of the algorithm used for the integration of the approach described in the previous chapter as a theory solver in the SMT-RAT-MCSAT framework. The experimental results of evaluating the performance of the new algorithm are discussed in **Chapter 6**. Finally, in **Chapter 7** we give a summary of this thesis and a short outline of possible future work.

The main contributions of this work are as follows:

- Implementation of the Fourier-Motzkin variable elimination in order to be able to explain why all extensions of a given partial theory assignment are inconsistent with a set of theory constraints.
- Integration of the former implementation as theory solver into the MCSAT framework of the SMT solver SMT-RAT.
- Analysis of the efficiency of the approach by comparing running times of standard SMT solvers for linear real arithmetic with running times of the SMT-RAT MCSAT approach with our implementation.

Chapter 2

Preliminaries

2.1 Technical background

As usual, we denote the ring of integers with $\mathbb{Z}$, the field of rational numbers with $\mathbb{Q}$, and the field of real numbers as $\mathbb{R}$. In all that follows, we assume a finite set of variables X ranging over $\mathbb{R}$. *Variables* in X are denoted by $x_1, x_2, ..., x_n$ and *coefficients* in $\mathbb{Z}$ by *a, b, c, d.*

All *(linear) polynomials p, q, r, s* over X with coefficients in $\mathbb{Z}$ are assumed to be a sum of monomials in the form $a_1x_1 + ... + a_nx_n + c$. In all what follows, these polynomials have its variables ranging over the field of $\mathbb{Q}$ instead of $\mathbb{R}$.

E is used to denote *equalities* $a_nx_n + ... + a_1x_1 + c = 0$, J is used to denote *inequalities* $a_nx_n + ... + a_1x_1 + c \leq 0$.

$\alpha(x)$ refers to the *current assignment* of a variable $x \in \mathbb{R}$. Φ and Ψ denote a *set of formulas*, and ϕ and ψ denote *single formulas.*

With $dom(x)$ we refer to the *domain* of the variable x.

Definition 2.1.1. *(Polynomial). A term of the form*

$$p = \sum_{k=1}^{m} M_k$$

with M_k referring to monomials, is called a polynomial.

2.2 Linear real arithmetic

The theory of *linear arithmetic* is a logic that allows inequalities and equations over real numbers.

Definition 2.2.1. *(Theory of Real Arithmetic).*

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

Considering a *polynomial* as the sum of *monomials*, a polynomial (see **Definition 2.1.2**) p over the reals consists of coefficients in $\mathbb{R}$, variables and the function symbols "$+$" and "$\cdot$". In all what follows, we only consider *linear polynomials* by only allowing multiplication with a constant. In this context, atoms are defined as polynomial constraints.

In this context, an example of atom will be:

$$x + y + 2z - 13 \leq 0$$

and an example of formula:

$$x \geq 0 \wedge (x + y \leq 2 \vee x - y \geq 6) \wedge (x + y \geq 1 \vee x - y \geq 4).$$

By $deg(p) := max_{1 \leq j \leq k} \sum_{i=1}^{n} e_{i,j}$ we denote the *degree of p*. The set of all polynomials with coefficients in $\mathbb{R}$ and variables x_1,...,x_n is denoted as $\mathbb{R}[x_1,...,x_n]$. We define *univariate polynomials* as those polynomials in $\mathbb{R}[x_i]$ for some variable x_i. A polynomial p of d variables and $deg(p) = n_1,...,n_d$ in $\mathbb{R}[x_1,...,x_n]$, which can be interpreted as a univariant polynomial in $\mathbb{R}[x_1,...,x_{n-1}][x_n]$, is called *multivariant polynomial*. These multivariant polynomials are then used to define the so called *quantifier-free linear arithmetic constraints*, being each of those constraints a polynomial compared to 0.

Definition 2.2.2. *(Polynomial constraint). An expression of the form*

$$p \bowtie 0,$$

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

In order to have a formal fundament for the LRA expressions in concerned, we introduce the *quantifier-free linear real arithmetic (QFLRA)* formulas.

In all that follows, we consider the satisfiability problem for the *quantifier-free fragment QFLRA*, or equivalently, the *existential fragment*. This means that there are no universal quantifiers and also, no negation of expressions containing existential quantifiers.

Definition 2.2.3. *(QFLRA formulas).*

QFLRA formulas are Boolean combinations of linear polynomial constraints.

The *syntax of a QFLRA formula* ϕ is shown in **Table 2.1.**

$t ::= 0$	$\mid$	1	$\mid$	x	$\mid$	$t + t$
$c ::= t$	$\mid$	$t < t$	$\mid$		$\mid$	
$\phi ::= c$	$\mid$	$\neg\phi$	$\mid$	$\phi \wedge \phi$	$\mid$	$\exists x.\phi$

Table 2.1: Syntax of a QFLRA formula

Example 2.2.1. *The following formula ϕ is a QFLRA formula with $dom(x)$ and $dom(y)$ being $\mathbb{R}$.*

$$\phi := \exists x.\ \exists y.\ x + 2y > 10 \wedge x \geq y \wedge (x < 0 \vee 2y > x)$$

A variable is *lower-bounded* (see **Definition 2.2.5**) by a constraint if its coefficient is positive, and *upper-bounded* if it is negative. An *interval* (see **Definition 2.2.4**) is a connected subset of the rational line $\mathbb{Q}$.

Definition 2.2.4. *(Intervals). An interval $I \in \mathbb{R}$ is a connected subset of reals. For a,$b \in \mathbb{R}$, we have two basic types of intervals:*

$$(a,b) = \{x \in \mathbb{R} | a < x < b\}$$
$$[a,b] = \{x \in \mathbb{R} | a \leq x \leq b\}.$$

a and b are the lower and upper bounds of the open (a,b) and closed [a,b] interval.

An interval $I = [a,b]$ with $a = b$ is called a point interval.

Definition 2.2.5. *(Bounds). A constraint of the form $x + a \bowtie 0$, with $a \in \mathbb{R}$, $\bowtie \in \{<, >, \leq, \geq, =\}$ and a variable x is called a bound.*

It is important to note that for the purpose of this thesis, the inputs are restricted for the fragment of the rational numbers $\mathbb{Q}$, that is, $\mathbb{Q} = \mathbb{R}$.

Chapter 3

Satisfiability checking

Problems in different areas like theorem proving, model checking, verification, synthesis, just to mention a few well-known examples, are effectively modeled in terms of logic. First of all, these problems need to be formalized. Then, it is necessary to check the *validity* and *satisfiability* of the formulas, and in case they are *satisfiable*, to identify satisfying solutions. The algorithms used through this process are known as *decision procedures*.

In this context, *satisfiability checking* is a line of research which aims in checking the satisfiability of existentially *quantified logical formulas*. There are different approaches which involve tools to check whether or not certain formulas are *satisfiable*, that is, to decide whether such a formula is *satisfiable*; these tools are known as *solvers*.

If we focus the problem of *satisfiability checking* on *boolean propositional logic*, we come to the concept of *SAT solvers*, which are powerful engines able to solve an impressively large set of propositional logic problems.

Due to the success of the *SAT solvers*, the *SAT-modulo-theories (SMT) solvers* started to being developed by the satisfiability checking community with the purpose of enriching the propositional SAT solvers with solver modules for different theories. Nowadays, these solvers are available for a wide range of theories like equalities and uninterpreted functions, bit-vector arithmetic, floating-point arithmetic, array theory, difference logic, (quantifier-free) linear real/integer/mixed arithmetic, and (quantifier-free) non-linear real/integer/mixed arithmetic.

3.1 Satisfiability (SAT) solvers

The *boolean satisfiability problem*, abbreviated as SAT, refers to the problem of determining whether there exists an assignment that satisfies a given *propositional formula*.

Definition 3.1.1. *(SAT).*

Given a (closed) logical formula ϕ in a decidable background theory τ, which constraints the interpretation of the symbols used in ϕ, SAT is the problem of deciding whether there exists a satisfying assignment to the free variables in ϕ, that is, answering the question of whether is a model of τ that makes ϕ true.

Example 3.1.1.

$$a = True,\ b = True,\ c = False$$

is a satisfying assignment for

$$(a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg c)$$

Definition 3.1.2. *(Propositional formula).*

A propositional formula ψ is built from a set of Boolean variables, the logical constants true and false, the logical connectives conjunction ($\vee$) , disjunction ($\wedge$), implication ($\subset$), bi-implication($\leftrightarrow$) and negation ($\neq$), and parenthesis.

Despite the *NP-completness* of SAT problems for propositional logic, which means that probably there is no algorithm able to solve it in polynomial-time, there are in practice some algorithms that perform well on many SAT instances even with this bad worst-case complexity. One of them is the *DPLL-style* algorithm (illustrated in **Figure 3.1**), which is implemented today in most state-of-the-art SAT solver technologies. It was introduced by Davis, Putman, Logemann and Loveland [DLL62] [DP60] and its basic principle is performing a case split on the truth values of variables by a *"backtracking depth-first search"*. Whenever the solver encounters a variable assignment in which one of the clauses of the formula evaluates to *false*, it backtracks and changes the most recent assignment until all assignments have been explored. Before this algorithm can be applied, the SAT instance needs to be transformed into *Conjunctive Normal Form (CNF)*. Every propositional formula ϕ can be transformed into an equisatisfiable formula ϕ' in CNF with only a linear growth in size using *Tseitin's transformation* [Tse83].

Definition 3.1.3. *(CNF). A formula ϕ of the form*

$$\phi = \wedge_{i=1}^{n} \vee_{j=1}^{m} p_{ij}$$

with literals p_{ij}, where a literal is either a proposition or its negation, is in CNF. The disjunctions of literals are called clauses.

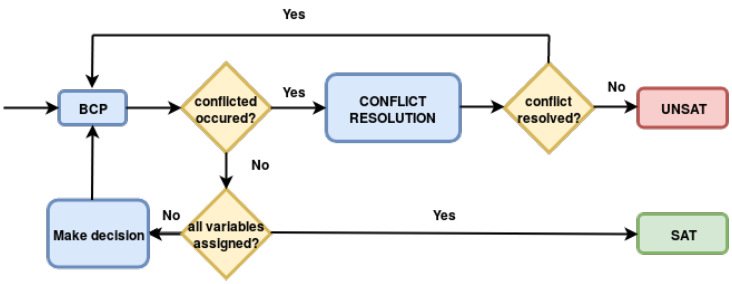

Figure 3.1: The DPLL framework.

There are also many further optimizations proposed after the DPLL algorithm, which led to major improvements, such as the *Conflict-Driven Clause Learning (CDCL)* in real-world problem domains.

3.2 Satisfiability-modulo-theories (SMT) solvers

The generalization of the SAT problem into first-order domain is called *Satisfiability Modulo Theories (SMT)*. SMT solvers extend SAT solvers with theories. While SAT checks formulas from the propositional logic for satisfiability, SMT combines a SAT solver with a theory solver to decide formulas from the existential fragment of first-order logic over some theory. In first-order logic, a model assigns values from a domain to variables and interpretations over the domain to the function and predicate symbols.

Definition 3.2.1. *(Theory). A theory is a set of formulas* Φ *closed under logical consequence, i.e. such that for any formula* ϕ *we have* $\Phi \models \phi$ *iff* $\phi \in \Phi$.

In comparison to SAT solving, the Boolean atoms represent now constraints over individual variables ranging over integers, reals, bit-vectors, datatypes, and arrays; and the constraints can involve theory operations, equality and inequality.

We can distinguish between *"less-lazy"* and *"full-lazy"* SMT solving. The former uses a SAT solver to find solutions of the Boolean skeleton of a SMT formula and invokes dedicated theory solvers to check the consistency in the underlying theory; the latter searches for a complete Boolean solution before invoking theory solvers. In this thesis only SMT solvers constructed according to the so-called *"less-lazy"* SMT solving approach, outlined in **Figure 3.2**, are considered.

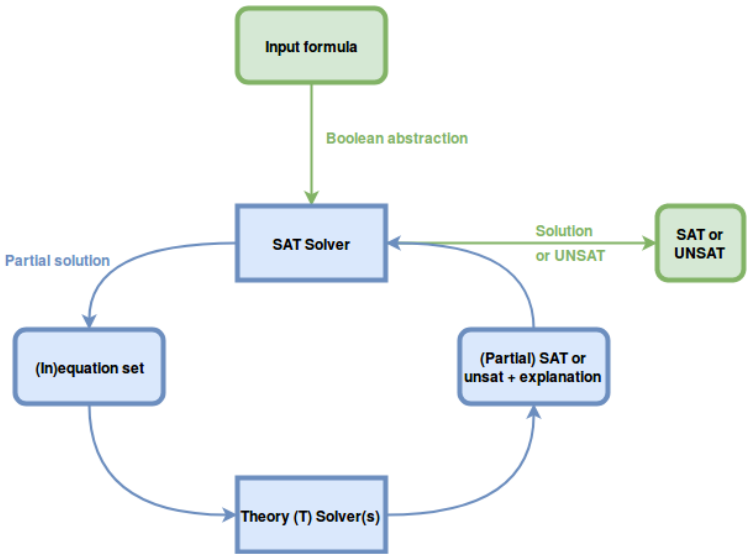

Figure 3.2: Typical SMT solver structure.

Less-lazy SMT solvers combine a *SAT solver* tightly integrated with a *T-solver*. Thereby the SAT solver handles the logical structure of the input formula and is responsible for finding solutions for the Boolean skeleton of such a formula. In order to being able to check the consistency of theory atoms, the SAT solver needs to communicate with the theory solvers, which are in charged of implementing decision procedures for the underlying theory. As mentioned in the previous section, the mayority of the SAT solvers uses the DPPL algorithm. The extension of the previous

algorithm to incorporate reasoning about an specific theory, which can be extended to combined theories, is called *DPLL(T)*.

The approach has several variants, differing in the sophistication of the interaction between the SAT engine and the theory solvers. Examples or solvers based on the "less-lazy approach" are CVC4, MathSAT, Z3 or SMT-RAT, which is described in a later section.

3.3 Quantifier Elimination Methods

Given a *quantifier* formula ϕ, *quantifier elimination* is the process of finding an equivalent, *quantifier-free* formula ϕ'. The possibility of applying a QE method in theory and practice in general depends on the considered formal system and the underlying theory.

Definition 3.3.1. *(Quantifier elimination). A theory T admits quantifier elimination if there is an algorithm that given an arbitrary T-formula φ, produces T-formula ϕ s.t.:*

- *ϕ is quantifier-free*
- *$\varphi \Leftrightarrow \phi$*

So if T admits quantifier elimination and the satisfiability problem of quantifier-free theory of T is decidable, then T is decidable.

We focus on input formulas found in the SMT setting with only existential quantifiers and no free variables; these formulas are called *quantifier-free formulas*.

3.3.1 Motivation of QE methods in SMT solving

One critical feature in the context of SMT solvers is to be able, from an unsatisfiable set of constraints, to extract a small conflict set. Without this ability, the SMT solver would enumerate an exponential number of slightly different assignments. All these assignments would be rejected by the solver, but they would essentially be unsatisfiable for the same reason. Hence we focus on the computation of small conflict sets from unsatisfiable sets of constraints. That is the reason we aim to use quantifier elimination methods. For *non-linear* arithmetic constraints, *cylindrical algebraic decomposition (CAD)* and *virtual substitution* are two commonly used real QE methods. *Fourier-Motzkin elimination* is one of the most well-know QE methods for linear real arithmetic.

3.4 A model-constructing satisfiability calculus

A compelling recent development in SMT solving is the *model-constructing satisfiability calculus (MCSAT)*, an extension of the DPLL(T) implemented in most of the current SMT solvers. The main difference between the previous frameworks is that MCSAT is not restricted to Boolean decisions. Alternatively, the model which the theory is trying to construct is involved in the search and in explaining the conflicts.

The way in which the satisfaction of the Boolean structure is assured is similar to how it is done in DPLL(T); however, in contrast to the standard SMT approach, the search for a propositional satisfying assignment is extended with a search for an assignment of values to the theory variables, that is, such an assignment is constructed simultaneously by "guessing" values for the theory variables. In case of *conflicting assignment*, the theory solver needs to *explain the conflict* of a set of constraints under a given partial assignment to the theory variables.

In comparison to the DPLL(T) framework, the *input* is a *partial assignment* in addition to the set of constraints whose consistency needs to be checked. The *states in the transition system* are pairs of the from $\langle M,C\rangle$, where M is a sequence or trail of trail elements, and C is a set of clauses. Each trail element is either a decided literal, a propagated literal, or a model assignment. Both decided literals and model assignments are referred as *decisions*. While a *decided literal* is a literal that is assumed to be true, a *propagated literal* ($C \rightarrow L$) marks a literal L that is implied to be true in the current state by the clause C (the explanation). A *model assignment* $x \mapsto \alpha$ is an assignment of a first-order uninterpreted symbol x to a value α.

3.4.1 Theory specific rules

Considering the *clausal rules* of abstract DPLL, now in MCSAT we have *theory specific rules* (see **Tables 3.1** and **3.2**), which extend DPLL rules to enable theory-specific reasoning, allowing deductions in the style of DPLL(T), but more flexible, and allowing for assignments of variables to particular concrete values.
As in DPLL(T), the basic requirement for a theory decision procedure is to provide an *explain* function, able to explain theory-specific propagations and infeasible states; though, such a explain function in MCSAT is more flexible. Given a literal L and a consistent trail M which implies L to be true, *explain(L,M)* must return a valid theory lemma $E = L_1 \vee ... L_k\ L$ (in DPLL(T) the theory explanations were theory lemmas in form of clauses that only contain negations of literals asserted so far). The literals of E must be from the finite basis $\mathbb{B}$, ensuring by this way the termination of the procedure; and all literals L_i must evaluate to *false* in M. By allowing explanations to contain more that just the literals of the trail allow for more expressive lemmas, which is crucial for model-based decision procedures.

RESOLVE			
$\langle [\![M, D \to L]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash R$	**if**	$\neg L \in C$, R =resolve(C,D,L)
CONSUME			
$\langle [\![M, D \to L]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash C$	**if**	$\neg L \notin C$,
$\langle [\![M, L]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash C$	**if**	$\neg L \notin C$,
BACKJUMP			
$\langle [\![M, N]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle [\![M, C \to L]\!], C \rangle$	**if**	$C = L_1 \wedge ... \wedge L_m \wedge L$ $\forall$: value(L_i,M) = $false$ value(L,M) = undef N starts with a decision
UNSAT			
$\langle M, \mathcal{C} \rangle \vdash$ false $\rightarrow$ unsat			
LEARN			
$\langle M, \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \cup \{C\} \rangle \vdash \mathcal{C}$	**if**	$C \notin \mathcal{C}$

Table 3.1: Clausal conflict analysis rules.

T-PROPAGATE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, E \to L]\!], \mathcal{C} \rangle$	**if**	$L \in \mathbb{B}$, value(L,M) = undef infeasible($[\![M, \neg L]\!]$) E = explain($[\![M, \neg L]\!]$)
T-DECIDE			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle [\![M, x \mapsto \alpha]\!], \mathcal{C}$	**if**	$x \in vars_T(\mathcal{C})$ $v[M](x)$ = undef consistent($[\![M, x \mapsto \alpha]\!]$)
T-CONFLICT			
$\langle M, \mathcal{C} \rangle$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash E$	**if**	infeasible(M) E = explain(false, M)
T-CONSUME			
$\langle [\![M, x \mapsto \alpha]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle M, \mathcal{C} \rangle \vdash C$	**if**	value(C,M) = false
T-BACKJUMP-DECIDE			
$\langle [\![M, x \mapsto \alpha, N]\!], \mathcal{C} \rangle \vdash C$	$\longrightarrow \langle [\![M, L]\!], \mathcal{C} [\![$	**if**	$C = L_1 \wedge ... \wedge L_m \wedge L$ $\exists i$: value(L_i,M) = undef value(L,M) = undef

Table 3.2: Theory search and conflict rules.

3.4.2 Producing Explanations

Assuming that the procedure has already assigned the variables x_1,...,$x_n \in Vars$ to real values from $\mathbb{R}$ ($\alpha : \{x_1,...,x_n\} \rightarrow \mathbb{R}$) and that it cannot find a theory value for a variable y that is consistent with a set D of constraints. Then D is called a set of *conflicting constraints* under the assignment α. Hence, given such a infeasible trail M, consisting of D and the assignment α, the framework must be able to produce a valid theory lemma, inconsistent with M, by using only literals from the finite basis.

In principle, for any theory that admits *quantifier elimination*, constructing an explanation function *explain* that satisfies the finite basis requirement is possible. The basic idea is to eliminate all unassigned variables and produce an implied formula that is also inconsistent with the assigned variables in the infeasible trail.

Being A the conjunction of the conflicting constraints

$$A \equiv \bigwedge_{l \in D} l$$

and B the literals under the assignment α, we use a quantifier elimination procedure to generate a CNF formula F of the form $C_1 \wedge ... C_k$ that is equivalent to $(\exists y : A)$, and therefore also inconsistent with B while only using the assigned variables (in this case, x). Hence, each explanation T needs to be valid and needs to exclude the current assignment; that is, it must hold that $A \wedge T \rightarrow \bigvee_{i=1,...,n} x_i \neq B$. Furthermore, all these constraints in the explanation must be from $\mathbb{B}$.

Definition 3.4.1. *(Explanation.) Assuming a set D of conflicting constraints containing variables x_1,...,x_n, y and an assignment $\alpha : x_1,...,x_n \rightarrow \mathbb{R}$ so that for each extension α' of α for y, α' is conflicting with $A \equiv \bigwedge_{l \in D} l$. Then explain(A,$\alpha$) is a theory lemma T so that*

- $\models T$,
- *T is of the form $A \rightarrow \phi$ for some formula ϕ where $\alpha \not\models \phi$ and*
- *$l \in \mathbb{B}$ for all constraints l in T*

3.4.3 Linear arithmetic

When we are solving a set of LRA $\mathcal{C}$, the *Fourier-Motzkin elimination* is sufficient to define the explain function, as shown in [MKS09], [KTV09]. FME gives a finite basis $\mathbb{B}$ with respect to $\mathcal{C}$, and the basis can be obtained by closing $\mathcal{C}$ under the application of FME step. Clearly, the closure is a finite set, since we always produce constraints with one variable less.

As we mentioned in the **Introduction**, there already exists an MCSAT approach for NLRA, but there is no one available for LRA. Since the *Simplex method* is not suitable for explanation, we appeal to the *Fourier-Motzkin method* as explanation generator.

Example 3.4.1. *(MCSAT explanation performance for unsatisfiability).*

Considering the set of LRA unit clauses:

$$\mathcal{C} = \{z \geq 1,\ z \leq x,\ x < 1,\ x < y\}$$

We start the deduction from the initial state $\langle [\![]\!],\ \mathcal{C}\rangle$ *and apply the rules from the MCSAT system shown in* ***Tables 3.1*** *and* ***3.2****. For simplicity, whenever the literal* L *is implied by the unit clause* L*, we use the denotation* $\hookrightarrow L$ *instead of* $L \to L$*.*

$\langle [\![]\!],\ \mathcal{C}\rangle$

$\downarrow$ *PROPAGATE* $\times$ *4 (all the unit clauses are propagated)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ \mathcal{C}\rangle$

$\downarrow$ *T-DECIDE (the current trail is consistent with the model assignment* $x \mapsto 0$*)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ x \mapsto 0,\ \mathcal{C}\rangle$

$\downarrow$ *T-DECIDE (peek a value for* y*, keeping the consistency;* y *s.t* $x < y$*)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ x \mapsto 0,\ y \mapsto 1,\ \mathcal{C}\rangle$

$\downarrow$ *T-CONFLICT (*$z \geq 1$ *and* $z \leq x$ *implies* $1 \leq x$*)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ x \mapsto 0,\ y \mapsto 1,\ \mathcal{C}\rangle \vdash C$

The conflict was detected by recognizing that we cannot pick a value for z*, because the trail contains* $z \geq 1$*,* $z \leq x$ *and* $x \mapsto 0$*. Hence, the explain function generates the explain clause*

$$C \equiv \neg(z \geq 1) \vee \neg(z \leq x) \vee 1 \leq x$$

by eliminating z *using the Fourier-Motzkin Elimination. The former clause evaluates to false in the current trail.*

The deduction is continued by analyzing the conflict.

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ x \mapsto 0,\ y \mapsto 1,\ \mathcal{C}\rangle \vdash C$

$\downarrow$ *T-CONSUME (the conflict does not depend on* y*)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ x \mapsto 0,\ \mathcal{C}\rangle \vdash C$

$\downarrow$ *BACKJUMP (after backtracking* $x \mapsto 0$*, the clause* $\mathcal{C}$ *implies* $1 \leq x$*)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ C \to 1 \leq x,\ \mathcal{C}\rangle$

After the application of the backjump rule, the lately asserted literal $1 \leq x$ *is immediately in conflict with the literal* $x < 1$ *and we enter again in conflict resolution by Fourier-Motzkin Elimination.*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ C \to 1 \leq x,\ \mathcal{C}\rangle$

$\downarrow$ *T-CONFLICT (*$1 \leq x$ *and* $x < 1$ *implies false)*

$\langle \hookrightarrow z \geq 1,\ \hookrightarrow z \leq x,\ \hookrightarrow x < 1,\ \hookrightarrow x < y,\ C \to 1 \leq x,\ \mathcal{C}\rangle \vdash$
$\neg(1 \leq x) \vee \neg(x < 1)$

$\downarrow$ *RESOLVE* $\times$ *3, CONSUME, RESOLVE, UNSAT*

3.5 SMT-RAT

SMT-RAT means *"Satisfiability-Modulo-Theories Real Arithmetic Toolbox"*. It is an open-source C++ toolbox for strategic and parallel SMT solving consisting of SMT compliant implementations of methods for solving quantifier-free first-order formulas; these methods are referred as modules. The former modules can be combined either to an SMT solver or a theory solver in order to extend the supported logics of an existing SMT solver by the supported logics of SMT-RAT.

SMT-RAT focuses on non-linear real and integer arithmetic, but it also supports theories as linear real and integer arithmetic, difference logic, bit-vectors and pseudo-Boolean constraints.

In comparison to other theory solvers, as mentioned in [CLJÁ12] the main advantages of this toolbox are:

1. A *complete* SMT-compliant decision procedure.
2. The possibility to *combine* theory solvers according to user-defined strategy.
3. A *modular* and *extendable open-source* implementation.

There are two strategies of SMT-RAT: *SMT-RAT* and *SMT-RAT-MCSAT*. For the purpose of this thesis, we are interested in the second one.

3.5.1 SMT-RAT-MCSAT

SMT-RAT-MCSAT uses a preliminary implementation of the MCSAT framework proposed by D. Jovanović and L. de Moura [DMJ13], equipped with an NLSAT-style CAD-based explanation function, complemented with a simpler explanation function based on the Fourier-Motzkin variable elimination. The general MCSAT framework is integrated in an adapted minisat solver [ES03]; thought, it is not optimized yet.

Chapter 4

The Fourier-Motzkin Elimination

The *Fourier-Motzkin Elimination (FME)* was the earliest method for solving *linear inequality systems*. It was discovered in 1826 by Joseph Fourier, and re-discovered in 1936 by Theodore Samuel Motzkin. In the context of *satisfiability checking*, the *Fourier-Motzkin Elimination* can be used to decide whether a given system of constraints over the reals is *satisfiable*.

Gauss' recursive method (GE) of successively eliminating variables using linear combination of rows accomplishes the task of determining whether or not a system of equalities has a solution. This latter task is reminiscent of determining the same thing for a system of inequalities, and it is done by the *Fourier-Motzkin Elimination*. Nonetheless, *FME* differs from *GE* in that each step in the elimination can greatly increase the number of inequalities in the remaining variables.

We consider we have a given *linear system of inequalities* in the form

$$Ax \leq b$$

where $A \in \mathbb{R}^{m,n}$ and $b \in \mathbb{R}^m$.

That is, if we write the system S in component form, we have

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{1n}x_n & \leq & b_2 \\ & & & & \vdots & & & & \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \ldots & + & a_{mn}x_n & \leq & b_m. \end{array}$$

The *FME* performs successive variable eliminations in a breadth-first manner generating additional constraints with fewer variables. First, the equality constraints are eliminated in the same way they are eliminated in a system of linear equations *(GE)*. Then the inequalities are eliminated by using *Projection*.

We look for "$(\exists x) : Ax \leq b$" based on eliminating the first unknown variable x_1 and then proceeding recursively with the rest of the variables. The solution set of the system $Ax \leq b$ is a *polyhedron* which we denoted by P, i.e.:

$$P = \{x \in \mathbb{R}^n : Ax \leq b\}.$$

Say we start eliminating from x_1, which gives an equivalent system $S^{'}$. Hence x_1 lies in a certain interval which is determined by $x_2, x_3,...,x_n$. The polyhedron defined by $S^{'}$ is the projection of P along the $x_1 - axis$, that is, into the space of the variables $x_2, x_3,...,x_n$. Then we may proceed similarly and eliminate x_2, x_3 etc. Eventually, a system $l \leq x_n \leq u$ might be obtained, where l is defined as *lower bound* and u is defined as *upper bound*.

For a better understanding we show in **Figure 4.1** a possible representation of the *polyhedron* which constitutes the solution set of the system $Ax \leq b$. In the following schema, we can consider that any constraint with a positive coefficient for i_k is a *lower bound*, and any constraint with a negative coefficient for i_k is an *upper bound*.

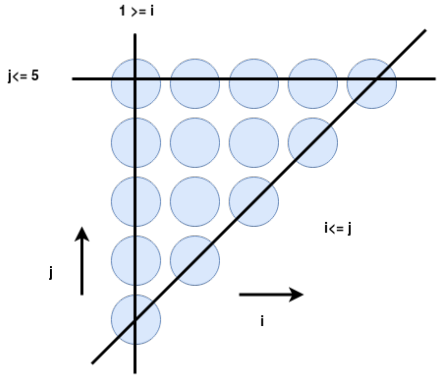

Figure 4.1: Polyhedron of the FM projection.

If $l > u$, it can be concluded that $Ax \leq b$ has no solution; otherwise, we may first choose a variable $x_n \in [l,u]$, and then choose x_{n-1} in an interval which depends on x_n, and so on. This back substitution procedure produces a solution $x = (x_1, x_2,...,x_n)$ to $Ax \leq b$, and every solution of $Ax \leq b$ may be produced in the same way. It is important to notice that if the system is inconsistent, this might possibly be discovered at an early stage and hence, the algorithm terminates.

Example 4.0.1. *(FM projection).*
Considering we have the set of constraints:

$$2x + y \leq 4 \tag{4.1}$$
$$x + y \geq 1 \tag{4.2}$$
$$y \leq 4 \tag{4.3}$$

If we eliminate y, by combining the constraint 4.1 with 4.2 and 4.3, the projection of ***Figure 4.2*** *is obtained.*

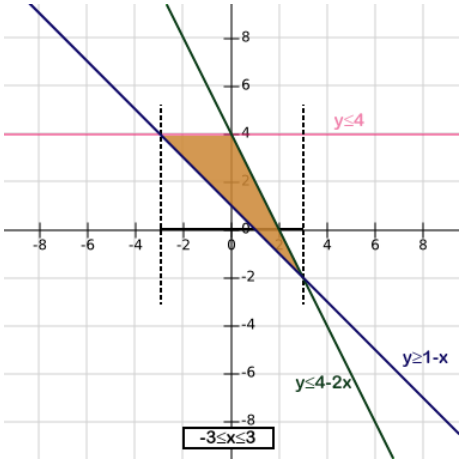

Figure 4.2: Example projection.

4.1 Algorithm description

The basic idea of the *variable elimination* procedure can be described in the two following steps:

1. Pick a variable and eliminate it, yielding an equisatisfiable formula that does not refer to the eliminated variable any more.
2. Continue until all the variables are eliminated.

The previous procedure is done by *collecting the requirements* for the *lower* and *upper bounds* on the variable we want to eliminate. Considering we have a formula $\phi(x_1, \ldots, x_n)$ containing only inequalities, a variable x_i is chosen to be eliminated and it is classified according to if it is an *upper bound*, a *lower bound* or it is *not a bound*. Once this is done, x_i is eliminated, forming $\phi'(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ s.t. *the original formula* $\phi(\ldots)$ *is satisfiable if and only if* $\phi'(\ldots)$ *is satisfiable*. The previous steps might be repeated until all the variables are not bounded anymore.

More in detail, the algorithm can be described as follows:

1. Choose to eliminate x_1 from $\phi(x_1,\ldots,x_n)$:

$$\begin{aligned} a_1 x_1 + \beta_1 &\leq 0 \\ &\vdots \\ a_m x_1 + \beta_m &\leq 0 \end{aligned}$$

   where $\beta_1 = a_{1,2}x_2 + \ldots + a_{1,n}x_n - b_1$ (equivalently for all x_n).

2. Rearrange the requirements:

$$\begin{aligned} a_i x_1 &\leq \beta_i \ \ for \ \ 1 \leq i \leq L \\ \beta_j &\leq a_j x_1 \ \ for \ \ L < j \leq U \\ \beta_k &\leq 0 \ \ for \ \ U < k \leq R \end{aligned}$$

where a_i,$a_j > 0$, and L refers to lower bounds, U to upper bounds and R to no bounds.

3. Combine each pair:

$$a_i x_1 \leq \beta_i \ \ for \ \ 1 \leq i \leq L$$
$$\beta_j \leq a_j x_1 \ \ for \ \ L < j \leq U$$

as $a_i\beta_j \leq a_i a_j x_1 \leq a_j\beta_i$ which is *satisfiable iff* $a_i\beta_j \leq a_j\beta_i$ is *satisfiable.*

After the elimination we have:

$$\phi^{'}(x_2,...,x_n): \bigwedge_{\substack{1 \leq i \leq L \\ L < j \leq U}} a_i\beta_j \leq a_j\beta_i \bigwedge_{U < k \leq R} \beta_k \leq 0$$

Hence, the algorithm removes the variables that are not bounded in both ways (and all the constraints that use them). We can conclude that for each pair of lower bound β_l and upper bound β_u, we have:

$$\beta_l \leq x_n \leq \beta_u$$

And for each such pair, we add the constraint:

$$\beta_l \leq \beta_u$$

The problem is trivial when there is no variable. For instance:

$\ldots \wedge 4 < 3 \wedge \ldots \Rightarrow \phi_0$ is *unsatisfiable.*

Definition 4.1.1. *(Classification of the variable bounds).*

For a variable x_n, we can partition the constraints according to the coefficients of x_n:

- $a_{in} = 0$ *: the constraint i puts no bound on x_n.*
- $a_{in} > 0$ *: the constraint i puts an upper bound on x_n.*
- $a_{in} < 0$ *: the constraint i puts a lower bound on x_n.*

$$\sum_{j=1}^{n} a_{ij} \cdot x_j \leq bi$$

$$=> a_{in} \cdot x_n \leq b_i - \sum_{j=1}^{n} a_{ij} \cdot x_j$$

$$\textbf{\textit{(A)}} \rightarrow x_n \leq \frac{b_i}{a_{in}} - \sum_{j=1}^{n} \frac{a_{ij}}{a_{in}} \cdot x_j => \textbf{\textit{UPPER BOUND}}$$

$$\textbf{\textit{(B)}} \rightarrow x_n \geq \frac{b_i}{a_{in}} - \sum_{j=1}^{n} \frac{a_{ij}}{a_{in}} \cdot x_j => \textbf{\textit{LOWER BOUND}}$$

Example 4.1.1. *(Bounds classification).*

Considering we eliminate from x_1, hence we look for the category of x_1:

- $x_1 - x_2 + 2x_3 \leq 0$: *x_1 is an upper bound.*
- $-x_1 + x_3 \leq 0$: *x_1 is a lower bound.*
- $-x_3 + x_2 \geq -2$: *x_1 is not a bound.*

Example 4.1.2. *(FME for satisfiability checking). Given the formula:*

$$\phi(x,y,z): \quad y < 1 \ \wedge \ z > 1 \ \wedge \ -3x + y > 2 \ \wedge \ x = z - y$$

we perform the FME for satisfiability checking as follows:

1. *Eliminate equations: We substitute "$x + y$" for z in $\phi(x,y,z)$:*

   $$x = z - y \ \rightarrow \ z = x + y$$

   Which leads to: $\phi'(x,y,z): \quad y < 1 \ \wedge \ x + y > 1 \ \wedge \ -3x + y > 2$

   $\phi'(x,y,z)$ is satisfiable iff $\phi(x,y,z)$ is satisfiable.

2. *First, we choose to eliminate from x:*

   $$3x < y - 2 \tag{4.4}$$
   $$-y + 1 < x \tag{4.5}$$
   $$y - 1 < 0 \tag{4.6}$$

   We can categorize the variable x : it puts an upper bound on the variable y in (4.2), a lower bound in (4.3) and no bound in (4.4).

   Now, we collect the condictions for upper and lower bounds:

   $-3y + 3 < 3x < y - 2$ is satisfiable iff $-3y + 3 < 3x < y - 2$ is satisfiable. That is, iff the constraint $-4y < -5$ is satisfiable.

   The last leads to: $\phi''(x,y,z): \quad -4y < -5 \ \wedge \ y < 1$

   $\phi''(x,y,z)$ is satisfiable iff $\phi'(x,y,z)$ is satisfiable.

3. *We choose to eliminate from y. We can see that it is not possible to categorize the variable since it is already not bounded, so this might be the last step in the elmination:*

   $$5 < 4y \tag{4.7}$$
   $$y < 1 \tag{4.8}$$

   $5 < 4y < 4$ is satisfiable iff the constraint $5 < 4$ is satisfiable.

   This leads to: $\phi'''(x,y,z): \quad 5 < 4$

   $\phi'''(x,y,z)$ is satisfiable iff $\phi''(x,y,z)$ is satisfiable.

   $\phi'''(x,y,z)$ is a contradiction. Hence, the formula $\phi(x,y,z)$ is unsatisfiable.

4.1.1 Special cases

We need to consider two special cases: one in which we *only* have *upper bounds*, and another one in which we *only* have *lower bounds*.

We start with the first case. When we have *no lower bounds*, we must proceed as follows:

$$a_i x_1 \leq \beta_i \ for \ 1 \leq i \leq L \qquad \beta_k \leq 0 \ for \ L < k \leq R$$

Then,

$$\phi^{'}(x_2,...,x_n) : \bigwedge_{m'' < k \leq m} \beta_k \leq 0$$

and $\phi(x_1,...,x_n)$ is *satisfiable* iff $\phi^{'}(x_2,...,x_n)$ is *satisfiable*, because x_1 can be chosen arbitrarly small.

The opposite case, when we have *no upper bounds*, is analogous. The only difference is that now x_1 can be chosen arbitrarly large.

4.1.2 Strict inequalities

The approach works also if we have both non-strict and strict inequalities. What we need to change is the following:

1. We distinguish between *strict* and *non-strict* lower and upper bounds, defined respectively by non-strcit inequalities.
2. For each pair of lower and upper bounds, if any of them is strict then we add the constraint $\beta_l < \beta_u$ instead of $\beta_l \leq \beta_u$.

4.1.3 Complexity of the algorithm

Each elimination step in the procedure above introduces in the worst case a quadratic number of new constraints, making the procedure double exponential. For this reason, the Fourier-Motzkin elimination is usually not practical for large set of constraints.

Chapter 5

Using Fourier-Motzkin Elimination for explanations in MCSAT

We aim to integrate the Fourier-Motzkin Elimination (FME) into the MCSAT framework of the solver SMT-RAT. To such purpose, we need an explanation function according to **Definition 3.4.1**.

As input we have a conjunction of conflicting constraints A and a partial model M which refers to a partial assignment α. It is known that $\exists y.\ A$ is conflicting with α. Hence, we look for a reason to explain the unsatisfiability, together with the partial model, where the explanation is only formulated in the assigned variables. Such conflicting constraints are LRA polynomial multivariant constraints (see **Definition 2.2.2**) of the form:

$$A := \begin{array}{c} \sum_{i=1}^{n} a_{1,i} x_i \bowtie_1 0 \\ \vdots \\ \sum_{j=1}^{n} a_{m,j} x_j \bowtie_m 0 \end{array}$$

where x_1,...x_n are *variables*, $a_{m,n}$ are the *coefficients* of those variables, and $\bowtie \in \{<, >, \leq, \geq, \neq, =\}$; that is, we have a system of conjoined linear inequalities with m *constraints* and m *variables* in $\mathbb{R}$.

Equalities and inequalities are special cases. In our implementation of FME, we handle inequalities, but we do not consider equalities. If unless one inequality in the input set is found, we use the already implemented explanation function of the SMT-RAT-MCSAT; that is, an NLSAT-style CAD-based explanation function, complemented

with a simpler explanation function based on the Fourier-Motzkin variable elimination. FME could actually deal with equalities, but it looks more reasonable to handle these equalities in a Gauss-like fashion. This is the reason for ignoring the equalities for now, but we defer it to possible future work.

The procedure (shown in **Algorithm 1**) can be sketched as follows: We might perform the FME in the set of conflicting constraints evaluated in the partial model. The elimination may be done from a pre-assigned variable. In this sense, if, for example, we are eliminating from the variable x, but the current variable we are evaluating is y, such a variable will be ignored, and consequently, also the constraint in which it appears. The elimination is done in terms of a *classification of bounds* as explained in **Chapter 5**. Thence, we look for a set of upper and lower bounds for the variable we are eliminating from in the form:

$$\begin{array}{c} l_1 \leq | \ \ | \leq u_1 \\ \cdot \ \ | \ \ | \ \ \cdot \\ \cdot \ \ |x| \ \ \cdot \\ \cdot \ \ | \ \ | \ \ \cdot \\ l_{k1} \leq | \ \ | \leq u_{k2} \end{array}$$

Once we have obtained the previous set, we make intervals by combining the lower and upper bounds, but considering inequalities as well, if it is the case; though, we only consider those intervals which are in conflict. Then we select one of the conflicting intervals and build from it the formula that is going to be returned as explanation for the unsatisfiability.

Algorithm 1 Algorithm for the explanation function

```
1:  function getExplanation(A, Variable var, Model M)
2:      for all c_i ∈ S do
3:          Evaluate c_i in M
4:          if Var_evaluated ≠ var then
5:              Ignore variable
6:          end if
7:      end for
8:      Divide S into S_upper(x), S_lower(x), S_ineq(s) and S_noBound
9:      for all upper ∈ S_upper(x) do
10:         for all lower ∈ S_upper(x) do
11:             Create intervals for all combinations of upper and lower
12:         end for
13:     end for
14:     Choose the best conflict
15:     Build formula from conflict
16:     return Explanation
17: end function
```

5.1 Elimination of bounded variables

As defined in the previous chapter, in order to perform the elimination, we need to classify the constraints according to the coefficients of the variable x_i chosen to make the elimination from. Since we have a partial model, the classification of bounds only needs to be performed once.

There are three possible classifications for making the partitioning, namely, *upper bound*, *lower bound* and *no bound* (see **Definition 5.1.1**). Though, we are only interested on the first two.

Before making the classification, we need to take into account that there different types of input constraints, depending on what the relation symbol $\bowtie$ refers to. According to which type such constraints are, the way in which we keep record of them may differ.

1. $\bowtie$ ***is*** **"** $>$ **"** ***or*** **"** $\geq$ **"** :

   Depending on the sign of the variable coefficient a_i in the constraint c_i:

   if $\boldsymbol{c_i > 0}$ then the constraint c_i puts an *upper* bound on the variable x_n; otherwise, it puts a *lower* bound.

2. $\bowtie$ ***is*** **"** $<$ **"** ***or*** **"** $\leq$ **"** :

   Depending on the sign of the variable coefficient a_i in the constraint c_i:

   if $\boldsymbol{c_i > 0}$ then the constraint c_i puts a *lower* bound on the variable x_n; otherwise, it puts an *upper* bound.

3. $\bowtie$ ***is*** **"** $=$ **"** :

   As we mentioned before, these constraints will be ignored because we are not considering in our implementation of the Fourier Motzkin elimination.

4. $\bowtie$ ***is*** **"** $\neq$ **"** :

   This is a special case; we do not classify the bound at this point, but we keep it in a different subset. It needs to be considered apart because it may restrict the form of the explanation returned.

If any of those cases occur, then we consider that the constraint makes no bound on the variable in concerned.

Example 5.1.1. *(Classification of bounds).*

- $1 \leq x$ *: Lower bound*
- $x \leq 0$ *: Upper bound*
- $x \neq 0$ *: Inequality*
- $x = 0$ *: Equality (not considered)*

5.2 Intervals generation and conflict determination

Given a set of lower bounds S_l, a set of upper bounds S_u and a set of inequalities S_i, we want our framework to return an explanation for the unsatisfiability built from the "best" pair of upper and lower bounds. For this reason, we need a structure for keeping all the possible combinations of conflicting bounds. With this aim, we use a *representation of bounds* based on *intervals* (see **Definition 2.2.3**). In addition, we need to consider the set of inequalities obtained in the previous step in order to determinate whether there is a conflict or not.

Since in the end the resulting explanation will be built from the original constraints that originated the selected bounds, we cannot lose such constraints associated to their respective bounds. The library of C++ provides *tuples*, objects that pack elements of -possibly- different types together in a single object. Therefore, we create a list of tuples, each of one saving the two original constraints referring to their respective lower or upper bound, as same as the corresponding interval. In addition, if there is an inequality which influenced in the conflict generation, it is also recorded in the C++ structure, since we will need as well to consider it when we reach the step of building the formula from the conflict to be returned as explanation.

Algorithm 2 Algorithm for the intervals generation and conflict determination

```
1: function generatePairs(S_l, S_u, S_i, S_comb)
2:     for all u_i ∈ S_u do
3:         for all l_i ∈ S_l do
4:             Normalize bounds
5:             Create interval_i
6:             if interval_i is a point interval then
7:                 for all ineq_i ∈ S_i do
8:                     if Some ineq ∈ S_i excludes the point interval then
9:                         interval_i is a conflict
10:                        Add l_i, u_i, ineq_i and interval_i to S_comb
11:                    else
12:                        interval_i is not a conflict
13:                    end if
14:                end for
15:            else
16:                if interval_i is empty then
17:                    Add l_i, u_i and ineq_i to S_comb
18:                else
19:                    interval_i is is not a conflict
20:                end if
21:            end if
22:        end for
23:    end for
24: end function
```

We consider that *there is a conflict* when *the interval generated by the pair of lower and upper bounds is empty.* This may occur due to two possible reasons:

1. The interval generated is directly empty.
2. The interval generated is a point interval and some inequality excludes such a single point, converting the point interval into an empty interval.

The entire before described process is illustrated in the **Algorithm 2**.

5.3 Selection of the "best" conflict

Once we have generated all the conflicting intervals, we need to order them via some criterion which allows us selecting one of them, since we can only return an explanation built from one of these conflicts.

We talked in the previous section about the "*selection of the best pair of upper and lower bounds.*" Since there is no known optimal criterion to determine which conflicting interval is better to be used for building the explanation, we start with a trial criterion, consisting of ordering the intervals according to its diameter. In addition, we leave as further optimization, ordering the intervals following other criterions, which might improve the performance of the method.

Let's explain how the ordering of the intervals is made. We have implemented a method that sorts the vector of "tuples" created in the previous step according to the diameter of the intervals saved in each tuple. In order to do that, we use a *lambda function*, which is a well-known function used compare two elements within the sorting algorithm; hence, this lambda function is invoked inside the body of our sorting method. Theoretically, the lambda function should return true if and only if the first argument should come before the second argument; in our case, our lambda function should return true when the diameter of the first comparing interval is smaller than the one of the second comparing interval, making it coming first inside the vector of tuples.

Once all the vector of tuples is ordered, we consider as the "best" conflict of our ordered vector, the first element of such a vector; thus, it will be the tuple whose interval has the smallest diameter. We will use that selected interval to build the formula which is going to be returned as explanation.

5.4 Explanation generation

Being S_{sB} the conflict selected in the previous step, we need now to build the formula from such a conflict that is going to be used for explaining a reason for the unsatisfiability. The **Algorithm 3** shows how the formula for making the explanation is built.

In the following sketched algorithm it can be seen that the constraint C2 is built using the upper bound and the inequality. However, the former is equal to building the constraint from

$$lower - ineq \neq 0$$

The reason for this is that the interval we are considering is a point interval, and thence, upper and lower have the same value.

Algorithm 3 Algorithm for the building of the conflicting formula

```
1: function BUILDFORMULA(S_sB, Variable var, Inequality ineq)
2:     Take upper and lower bounds from S_sB
3:     Normalize upper and lower
4:     if inequality then
5:         Normalize ineq
6:         Make constraint C1 = upper − lower < 0
7:         Make constraint C2 = upper − ineq ≠ 0
8:         return res = (¬lower OR ¬upper OR ¬ineq OR C1 OR C2)
9:     end if
10:    Determine whether the resulting constraint is strict
11:    if Strict then
12:        resultingConstraint C = upper − lower < 0
13:    else
14:        resultingConstraint C = upper − lower ≤ 0
15:    end if
16:    return (res = ¬lower OR ¬upper OR C)
17: end function
```

As we said in **Chapter 4**, after each elimination we have an expression of the form:

$$\phi^{'}(x_2,...,x_n): \bigwedge_{\substack{1 \leq i \leq L \\ L < j \leq U}} a_i\beta_j \leq a_j\beta_i \bigwedge_{U < k \leq R} \beta_k \leq 0$$

In the previous formula, the variable x_1 has been already eliminated. Since the elimination is only performed once, the formula F from which we make the explanation is built now. Depending on the constraints that constitute the conflicting set, we can distinguish two cases.

- **Case 1: Without inequalities:** the conflict found comes from the case in which the interval formed by the lower and upper bounds in concerned is empty.

  The formula is built from the constraint

  $$\beta_l \bowtie \beta_u$$

  where β_l and β_u are the respectively lower and upper bound of the variable in concerned and $\bowtie$ can be either $\leq$ or $<$; and the tautology of the upper and lower bounds. That is:

  $$F \equiv (\neg\, \beta_l) \vee (\neg\, \beta_u) \vee (\beta_l - \beta_u \bowtie 0)$$

Example 5.4.1. *(Algorithm performance without inequalities).*

$$A := \{y \geq 2,\ y \leq x,\ x < 2\}$$

Hence, we have two variables {x,y}. Considering we are performing the elimination from the variable y, and our partial model is M := x = 1, the explanation generation by our FME implementation can be seen as follows:

1. *Considering the constraints:*
   - $\boldsymbol{y \geq 2}$ → *Evaluated in M:* $\boldsymbol{y \geq 2}$ → *It defines a* ***lower bound****.*
   - $\boldsymbol{y \leq x}$ → *Evaluated in M:* $\boldsymbol{y \leq 1}$ → *It defines an* ***upper bound****.*
   - $\boldsymbol{x < 2}$ → *Evaluated in M:* $\boldsymbol{0 = 0}$ → *It does not define any bound.*
2. *The following sets [(orginal constraint), (evaluated constraint)] have been found:*
   - $\boldsymbol{S_u} := [(y \leq x), (y \leq 1)]$
   - $\boldsymbol{S_l} := [(y \geq 2), (y \geq 2)]$
3. *The interval (0,0) is obtained from the combination of*

   $$[(y \leq x), (y \leq 1)] \text{ and } [(y \geq 2), (y \geq 2)]$$

   Here the interval obtained is the empty interval because the upper bound has a value of 1 and the lower a value of 2; that is, we have an overlapping of bounds, and consequently, a conflict.
4. *In this example the step corresponding to the ordering of the conflits and the selection of the best of them does not need to be done since we only have one element in each set of constraint.*
5. *The formula for the explanation is built from*

   $$(y \geq 2) \text{ and } (y \leq 0)$$

6. *The final explanation for the unsatisfiability is*

   $$(y - 2 < 0) \vee (-y + x < 0) \vee (-x + 2 \leq 0)$$

- **Case 2: With inequalities:** the conflict is generated because the interval was a point interval, and an inequality which excludes that single point was found, making an empty interval.

  The formula construction differs in:

  - We have $\beta_l \leq x_n \leq \beta_u$ and $ineq_n \neq x_n$
  - Then in the partial model we have $\beta_l = \beta_u = ineq_n$

  Hence the formula is now built from:

  $$F \equiv (\neg\, \beta_l) \vee (\neg\, \beta_u) \vee (\neg\, ineq_n) \vee (\beta_l - \beta_u < 0) \vee (\beta_l \neq ineq_n)$$

Example 5.4.2. *(Algorithm performance with inequalities).*

$$A := \{(y \geq x),\ (y \leq x)\ ,\ (y \neq x)\}$$

Hence, we have two variables $\{x,y\}$*. Considering we are performing the elimination from the variable y, and our partial model is* $M := x = 0$*, the explanation generation by our FME implementation can be seen as follows:*

1. *Considering the constraints:*
   - $\boldsymbol{y \geq x}$ → *Evaluated in M:* $\boldsymbol{y \geq 0}$ → *It defines a* ***lower bound****.*
   - $\boldsymbol{y \leq x}$ → *Evaluated in M:* $\boldsymbol{y \leq 0}$ → *It defines an* ***upper bound****.*
   - $\boldsymbol{y \neq x}$ → *Evaluated in M:* $\boldsymbol{y \neq 0}$ → *It is an* ***inequality****.*
2. *The following sets* $[(orginal\ constraint),\ (evaluated\ constraint)]$ *have been found:*
   - $\boldsymbol{S_u} := [(y \leq x), (y \leq 0)]$
   - $\boldsymbol{S_l} := [(y \geq x), (y \geq 0)]$
   - $\boldsymbol{S_{ineq}} := [(y \neq x),(y \neq 0)]$
3. *The interval* [0,0] *is obtained from the combination of* $[(y \leq x), (y \leq 0)]$ *and* $[(y \geq x), (y \geq 0)]$ *.*

   The interval is a point interval, and there in an inequality that excludes the point. Hence, an special conflict taking into account the inequality is built.
4. *As in the previous example, the step corresponding to the ordering of the conflits and the selection of the best of them does not need to be done since we only have one element in each set of constraint.*
5. *The formula from the explanation is built from*

   $$(y \geq x),\ (y \leq x)\ and\ (y \neq x)$$

6. *The final explanation for the unsatisfiability is*

   $$(y + x < 0) \vee \vee (-y - x < 0) \vee (y + x = 0)$$

Chapter 6

Experimental results

In order to analyze the efficiency of the approach presented in **Chapter 5** we compare the performance on benchmarks of the SMT-RAT-MCSAT framework with the Fourier-Motzkin elimination and the NLSAT-style as explanation functions with:

- The preliminary implementation of the MCSAT framework in SMT-RAT-MCSAT, which uses an NLSAT-style CAD-based explanation function.
- The regular SMT-RAT solver with CAD.
- The regular SMT-RAT solver with Simplex.
- The standard z3 solver.

As input problems we use the set of $QFLRA$ benchmarks provided by the SMT-LIB, and we establish a time limit of 30 seconds. After executing the benchmarks, different outputs can be obtained:

- *SAT or UNSAT.* The problem was correctly solved and is either satisfiable or unsatisfiable.
- *Resource exhaustion: timeout, memout or segfault.* The solver exceeds the time limit or the available memory before finding a solution. A *segfault* is an error which causes the program to crash due to an incorrectly accessing memory.
- *Wrong.* The solver computed a result, but it was not the current solution.

6.1 General performance

We start comparing the different outputs obtained after executing the benchmarks on the different satisfiability checking approaches. First of all, it is important to say that no wrongs occur in any of the approaches.

We focus on the relation between the number of solved tests and the number of test which did not finish as a consequence of a resource exhaustion, that is, a solution for the problem was not found by the solver before exceeding the 30 seconds time limit or the memory available.

We show in the **Table 6.1** the different benchmark results focusing on the ones referring to SAT or UNSAT, a resource exhaustion and segfaults. Then, in **Figure 6.1** we show the relation between the number of correctly solved tests, that is, those whose result is SAT or UNSAT, and the number of tests that did not finish due to a resource exhaustion or a segfault.

	SAT	UNSAT	TIMEOUT	MEMOUT or SEGFAULT
MCSAT+FM+CAD	376	283	988	2
MCSAT+CAD	370	278	998	3
Regular SMT + CAD	346	238	1064	1
Regular SMT + Simplex	455	356	835	3
z3	655	473	521	0

Table 6.1: Results on 1649 QFLRA benchmarks.

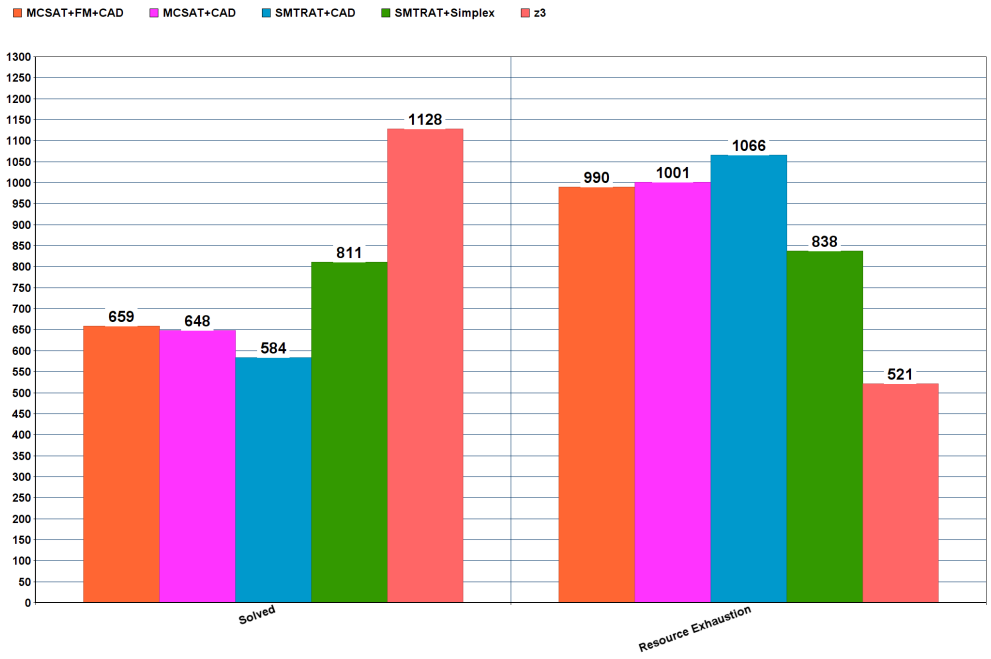

Figure 6.1: Graph bar comparing the relation between the number of correctly solved tests and the number of unfinished tests.

Ordering from highest to lower, we can say that the *number of solved tests* is

$$z3 > SMTRAT + Simplex > MCSAT + FM + CAD > MCSAT + CAD > SMTRAT + CAD$$

and the *number of tests which ended because of a resource exhaustion* is

$$SMTRAT + CAD > MCSAT + CAD > SMTRAT + FM + CAD > SMTRAT + Simplex > z3$$

Hence, our implementation has a better performance than the previous SMT-RAT-MCSAT using only CAD explanations and the regular SMT-RAT solver with CAD. It solves more instances and has less timeouts.

6.2 Running time performance on individual instances

To get a visual impression of how the solvers perform on individual instances, we include the **Figures 6.3 and 6.4**. We have already shown that SMT-RAT with Simplex and z3 are faster than any of the MCSAT approaches and SMT-RAT with CAD. Hence, we show the comparison of the running times of MCSAT-SMTRAT with FME and CAD explanations to the preliminary implementation of MCSAT-SMT-RAT with only CAD explanations in **Figure 6.3**, and to the SMT-RAT with CAD approach in **Figure 6.4**, on the benchmark sets.

On the plots, each point represents an instance of the benchmark set, its x coordinate represents the running time of our implementation and the coordinate y the running time of one of the other solvers. Hence, our approach with the FME is faster in all the instances that reside above the diagonal line.

We are not interested on the single points in the top right corner, because they correspond to instances where both solvers produce a timeout. The instances in which both solvers are fast create an opaque cloud close to the origin.

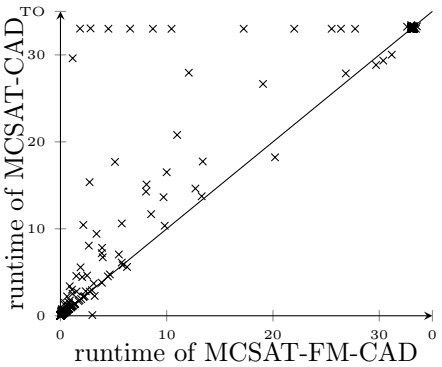

Figure 6.2: Comparison of running times between MCSAT-SMT-RAT with FME and CAD and MCSAT-SMT-RAT with CAD.

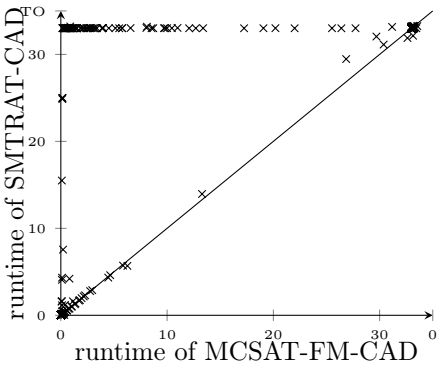

Figure 6.3: Comparison of running times between MCSAT-SMT-RAT with FME and CAD and SMT-RAT with CAD.

We can see in both plots that the new approach is faster than both the previous implementation of MCSAT-SMTRAT and the SMT-RAT with CAD explanations approach. Furthermore, we expect MCSAT with FME to be even better when equalities are considered.

6.3 Final analysis

In conclusion, in terms of the *time needed to find a solution*, we can say that our implementation of the Fourier-Motzkin elimination as explanation generator in the SMT-RAT-MCSAT framework has a better performance than the previously implemented approach for MCSAT in the SMT-RAT solver with an NLSAT-style and a CAD-based explanation function. However, our implementation does not perform better than the regular SMT-RAT approaches with Simplex and z3.

In terms of the *impossibility to solve a test due to a resource exhaustion*, our implementation performs a little better than the original implementation of the SMT-RAT-MCSAT, since it has less timeouts and more solved tests. We perform much more better than the regular SMTRAT with CAD explanations. However, the regular SMRAT with Simplex and z3 still perform better .

Chapter 7

Conclusion

7.1 Summary

This thesis started with an introduction to the concept of *satisfiability checking*, presenting both SAT and SMT solvers. We presented the SMT-RAT solver, since it was in its framework where the implementation of the Fourier-Motzkin elimination as explanation generator of the MCSAT approach was implemented. We came out as well with the basis of the MCSAT approach and the Fourier-Motzkin elimination algorithm.

In the following we focused in the main aim of this thesis, that is, how the FME was implemented in the SMT-RAT-MCSAT framework. We continued making an analysis of the performance of our implementation, by comparing it with other four techniques in terms of the time needed to find a satisfiable or unsatisfiable solution, and based on the relationship between the number of correctly solved tests and the number of tests which did not finished due to a resource exhaustion.

It turned out that we improve the performance of the original SMT-RAT-MCSAT and regular SMT-RAT with CAD explanations approaches in what refers to the number of correclty solved tests and in terms of the running time. In none of the cases we perform better than the regular SMT-RAT with the Simplex algorithm and z3.

7.2 Future work

There are some options to continue this work, most of them already mentioned in previous chapters.

Using Gauss elimination algorithm to deal with equalities. The original Fourier-Motzkin algorithm, as we presented in **Chapter 5** deals with equalities: the first step to be done in order to perform the algorithm is "eliminating the equality constraints in the same way they are eliminated in a system of linear equations with the Gauss elimination algorithm". However, in our original implementation we did not deal with equalities, since it can not be done by projection as we do with inequalities. When a problem with equalities is presented as input, we call the original implementation of the SMT-RAT-MCSAT framework. Hence, one improvement

to our approach would be dealing with equalities by handling them in a Gauss-like fashion.

Extension to Non-Linear Arithmetic. In some cases, to prove the satisfiability of polynomial problems, Linear Arithmetic reasoning is used as well. Hence, one approach to our work is expand our theory solver so it can expand polynomials and consider all monomial as independent variables.

Choosing a different criterion to order the conflicts. As we mentioned in the previous chapter, we do not have a specific criterion to choose the "best" conflict once all of them have been encountered. Hence, one possible extension is trying different criterions of selecting such conflict, and analyzing and comparing the performance of each of them.

Bibliography

[BCC+12] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A simplex-based extension of fourier-motzkin for solving linear integer arithmetic. In *IJCAR*, pages 67–81, 2012.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[CJ12] Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Smt-rat: An open source c++ toolbox for strategic and parallel smt solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

[CLJÁ12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. Smt-rat: an smt-compliant nonlinear real arithmetic toolbox. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 442–448. Springer, 2012.

[Dah07] Geir Dahl. Combinatorial properties of fourier-motzkin elimination. *Electronic Journal of Linear Algebra*, 16(1):29, 2007.

[DDM06] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for dpll (t). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006.

[DKPW10] Vijay DâĂŹSilva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 129–145. Springer, 2010.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[DMJ13] Leonardo De Moura and Dejan Jovanović. A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12. Springer, 2013.

[Dol98] Andreas Dolzmann. Solving geometric problems with real quantifier elimination. In *International Workshop on Automated Deduction in Geometry*, pages 14–29. Springer, 1998.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[Duf74] Richard J Duffin. On fourierâĂŹs analysis of linear inequality systems. In *Pivoting and Extension*, pages 71–95. Springer, 1974.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[GKS+16] Vladimir P Gerdt, Wolfram Koepf, Werner M Seiler, Evgenii V Vorozhtsov, et al. *Computer algebra in scientific computing*. Springer, 2016.

[JDF15] Maximilian Jaroschek, Pablo Federico Dobal, and Pascal Fontaine. Adapting real quantifier elimination methods for conflict set computation. In *International Symposium on Frontiers of Combining Systems*, pages 151–166. Springer, 2015.

[JDM11] Dejan Jovanović and Leonardo De Moura. Cutting to the chase solving linear integer arithmetic. In *International Conference on Automated Deduction*, pages 338–353. Springer, 2011.

[JLC] Sebastian Junges, Ulrich Loup, and Florian Corzilius. On gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers (extended version).

[KTV09] Konstantin Korovin, Nestan Tsiskaridze, and Andrei Voronkov. Conflict resolution. In *International Conference on Principles and Practice of Constraint Programming*, pages 509–523. Springer, 2009.

[MKS09] Kenneth L McMillan, Andreas Kuehlmann, and Mooly Sagiv. Generalizing dpll to richer logics. In *International Conference on Computer Aided Verification*, pages 462–476. Springer, 2009.

[Mon08] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 243–257. Springer, 2008.

[PS16] Martin Pfeifhofer and Felix Schett. Propositions as types. 2016.

[RHT+10] Andrew Reynolds, Liana Hadarean, Cesare Tinelli, Yeting Ge, Aaron Stump, and Clark Barrett. Comparing proof systems for linear real arithmetic with lfsc. In *Workshop on satisfiability modulo theories*. Citeseer, 2010.

[RS04] Harald Rueß and Natarajan Shankar. Solving linear arithmetic constraints. In *SRI International, Computer Science Laboratory*, 2004.

[RV07] Kristin Y Rozier and Moshe Y Vardi. Ltl satisfiability checking. In *International SPIN Workshop on Model Checking of Software*, pages 149–167. Springer, 2007.

[Str02] Ofer Strichman. On solving presburger and linear arithmetic with sat. In *International Conference on Formal Methods in Computer-Aided Design*, pages 160–170. Springer, 2002.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[Wil76] H Paul Williams. Fourier-motzkin elimination extension to integer programming problems. *Journal of combinatorial theory, series A*, 21(1):118–123, 1976.