MASTER OF SCIENCE THESIS

INCREMENTAL LINEARIZATION FOR SAT MODULO REAL ARITHMETIC SOLVING

Aklima Zaman

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

Advisor:
Gereon Kremer

Aachen, April 23, 2019

Abstract

Satisfiability Modulo Theories (SMT) solving is a technology for checking the satisfiability of quantifier-free first-order logic formulas over some theories. In this thesis, we implement the method which was proposed by the authors of [CGI+18]. The idea of the authors is to perform over-approximative linear abstraction iteratively on a non-linear formula by uninterpreted functions and check the satisfiability of the abstraction. If the abstraction is unsatisfiable then the original formula is unsatisfiable, too. Otherwise, if the solution for the linear formula does not satisfy the non-linear formula, then this linear formula is incrementally refined by a set of axioms. In this thesis we adapt this method to use linear abstraction by variables instead of uninterpreted functions and experiment with some heuristics aspects in SMT-RAT. We also propose a new axiom named Interval Constraint Propagation (ICP) axiom for refinement.

Eidesstattliche Versicherung

Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Acknowledgements

I am very thankful for the opportunity to work on this topic. My special thanks to Prof. Dr. Erika Ábrahám for her valuable and constructive suggestions during the planning and development of this thesis. Also thanks to the examiner Prof. Dr. Jürgen Giesl. I would also like to thank Gereon Kremer for his valuable advice and technical support on this thesis. Finally, I must express my gratitude to my parents and to my spouse for providing me continuous encouragement throughout the process of implementing and writing this thesis.

Contents

Chapter 1

Introduction

This thesis is about checking the satisfiability of logical formulas. If the logical formula is a quantifier-free non-linear real-arithmetic formula, then checking its satisfiability is computationally costly. The idea in the Ph.D. thesis [Irf18] and the related publication [CGI$^+$18] is to apply linear approximations which are over-approximative. For a given non-linear formula, the authors define a linear formula whose solution set over-approximates the set of satisfying solutions of the non-linear formula. If a solution for the linear formula is found in the over-approximation which does not satisfy the non-linear one, then they refine the linear abstraction by making the solution set smaller. The solution set gets smaller means that it is getting closer to the non-linear formula though in general it is not necessarily reached. That is why this method is incomplete. So, the authors of [CGI$^+$18] proposed an approach, referred to as Incremental Linearization, that trades the use of expensive solvers for non-linear formulas for an abstraction-refinement loop on top of computationally much less expensive solvers for linear formulas [CGI$^+$18].
Our contribution in this thesis is the implementation of this method in the SMT solver SMT-RAT and the experimental evaluation of different heuristics which are not addressed in the Ph.D. thesis [Irf18]. We have also contributed in this thesis work by introducing a new axiom referred to as the Interval Constraint Propagation (ICP) axiom as an additional axiom type based on which the refinement is performed.
The organization of this thesis is as follows. In Chapter 2 we introduce basic terms and definitions including a brief explanation on Incremental Linearization for non-linear real-arithmetic formulas as proposed in the Ph.D. thesis [Irf18]. Chapter 3 explains our contributions with algorithmic descriptions and examples. Chapter 4 describes the implementation. Chapter 5 discusses experimental results. Finally, we conclude the thesis in Chapter 6 and discuss possible directions for future work.

Chapter 2

Preliminaries

2.1 Propositional Logic

Propositional logic deals with logical relationships between propositions. A proposition is a boolean variable that has either a truth value *true* or a truth value *false*, also called atomic proposition. The syntax of well-formed propositional logic formulas is as follows:

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

where a is an atomic proposition. So, a propositional logic formula can be an atomic proposition or it can be constructed from atomic propositions by using logical connectives for negation $\neg$ and for conjunction $\wedge$. In the following we sometimes omit parentheses assuming that negation binds stronger than conjunction. There are some more logical connectives used in propositional logic, but they all are syntactic sugar. In the syntax, we have only $\neg$ and $\wedge$ as logical connectives because these are sufficient to express syntactic sugar. Syntactic sugar are:

$$\begin{array}{ccc}
\textit{false} & := & (a \wedge \neg a) \\
\textit{true} & := & (a \vee \neg a) \\
(\varphi_1 \vee \varphi_2) & := & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\
(\varphi_1 \rightarrow \varphi_2) & := & (\neg\varphi_1 \vee \varphi_2) \\
(\varphi_1 \leftrightarrow \varphi_2) & := & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
(\varphi_1 \oplus \varphi_2) & := & (\varphi_1 \leftrightarrow \neg\varphi_2)
\end{array}$$

Example 2.1.1. *Some propositional logic formulas are:*

$$\neg a$$

$$(\neg a \wedge \underbrace{(b \vee c)}_{\varphi_1})$$

$$(b \rightarrow \underbrace{(a \wedge c)}_{\varphi_2})$$

Here, a, b and c are atomic propositions, whereas φ_1 and φ_2 are propositional formulas.

2.2 First-Order Logic

Sometimes propositional logic is not enough for modeling and we need a more expressive language.
First-order (FO) logic is a framework which has the ingredients, theory symbols, predicate symbols and logical symbols. Theory symbols can be constants, variables, function symbols. All constants and variables are theory expressions. If $t_1, \ldots, t_n$ are theory expressions and f is an n-ary funtion symbol, then $f(t_1, \ldots, t_n)$ is also a theory expression. The difference between predicate symbols and function symbols is that the predicate symbols always give boolean values, but the function symbols give values from a theory domain. Predicate symbols are special function symbols, but they have boolean values and they made the switch from theory expressions to boolean expressions. If $t_1, \ldots, t_n$ are theory expressions and P is an n-ary predicate symbol, then $P(t_1, \ldots, t_n)$ is called a constraint. Logical symbols are logical connectives $\neg, \wedge, \vee, \rightarrow, \ldots$ and quantifiers. Two types of quantifiers are available, existential quantifier ($\exists$) and universal quantifier ($\forall$).
FO logic is called first-order logic because it allows the quantification over variables and it does not allow the quantification over predicates. We can have a hierarchy of logic, for example, second-order logic, third-order logic and so on. We are not going into the details of those as we are only interested in FO logic.

Example 2.2.1. *Assume the following:*

- *All girls are cute.*
- *Lily is a girl.*
- *Therefore, Lily is cute.*

We can formalize it by defining:

- *Constants:* *Lily*
- *Variables:* x
- *Predicate symbols:* *isGirl(·), isCute(·)*

Formalization:

- $\forall x.$ *isGirl(x)* $\rightarrow$ *isCute(x)*
- *isGirl(Lily)*
- *isCute(Lily)*

FO logic formulas can be constructed by the following syntax, where c is a constraint:

$$\varphi \quad := \quad c \quad | \quad \neg\varphi \quad | \quad \varphi \wedge \varphi \quad | \quad \exists x.\varphi \quad | \quad \forall x.\varphi$$

For $\forall x.\varphi$ and $\exists x.\varphi$, we call φ the scope of the quantification for x and we call occurrences of x in φ bound. An occurrence of a variable in a first-order logic formula is free if it is not bound. A formula without any free variable occurrence is called a sentence. A theory T is a set of sentences.

Example 2.2.2. *The following is a FO logic formula over the theory of integers with addition, also called linear integer arithmetic:*

$$\forall x. \exists y.\ (x+1<y) \ \wedge \ (x+2=z)$$

Here, 1 *and* 2 *are constants,* x, y *and* z *are variables,* $+$ *is a function symbol,* $<$ *and* $=$ *are predicate symbols,* $\wedge$ *is a logical connective and* $\forall$ *and* $\exists$ *are quantifiers. All occurrences of* x *and* y *are bound, whereas the only occurrence of* z *is free.*

2.2.1 Quantifier-Free FO Logic

A quantifier-free formula in FO logic is a formula that does not contain quantifiers. In other words, a quantifier-free formula is either a constraint which has truth values or the application of logical connectives to constraints. So, quantifier-free FO logic formulas have the following syntax:

$$\varphi \quad := \quad c \quad | \ \neg\varphi \quad | \ \varphi \wedge \varphi$$

Example 2.2.3. *An example quantifier-free FO logic formula for non-linear integer arithmetic including multiplication:*

$$((x+1<y) \ \wedge \ (x+2=z)) \ \vee \ (x+6=y*z)$$

If each constraint of a quantifier-free FO logic formula is replaced by a fresh boolean variable, then we get the boolean skeleton and the formula will become a propositional logic formula. To specify propositional logic as a quantifier-free first-order logic instance we only need boolean variables as the logical connectives are already in quantifier-free FO logic. Thus, propositional logic is also a FO logic.

Example 2.2.4. *The followings are examples for a quantifier-free FO logic formula* φ_{QF} *and its boolean skeleton* φ_{sk} *which is a propositional logic formula:*

$$\begin{array}{lccccccc} \varphi_{QF} & := & ((x+1<y) & \wedge & (x+2=z)) & \vee & (x+6=y*z) \\ \varphi_{sk} & := & (a & \wedge & b) & \vee & c \end{array}$$

2.3 Real Arithmetic

(Non-linear) real arithmetic (NRA) is a first-order logic over the theory of the reals with addition and multiplication. NRA constraints can be built upon constants r from some coefficient ring R (here we will use the rationals $\mathbb{Q}$), real-valued variables x and the functions addition $+$ and multiplication $*$ according to the following quantifier-free syntax:

$$\begin{array}{lcccccccccc} \text{Polynomials:} & p & := & r & | & x & | & p+p & | & p*p \\ \text{Constraints:} & c & := & p<p \\ \text{Formulas:} & \varphi & := & c & | & \neg\varphi & | & \varphi\wedge\varphi \end{array}$$

There are also syntactic sugar such as the relational operators $>, \leq, \geq, =, \neq$ and the logical connectives $\vee, \rightarrow$ etc.. We use μ to denote assignments (i.e., functions assigning

values to all variables) and write $\mu(x)$ for the value of x under μ. The semantics is defined as usual.
A monomial is a product of variables and the empty product represents the constant 1. A term $r * m$ is a product of a coefficient $r \in R$ and a monomial m. A polynomial is a theory expression. Polynomials are often written as sums of terms

$$p(x_1, \dots, x_n) = a_d m_d + \dots + a_0 m_0$$

with coefficients $a_d, \dots, a_0 \in R \setminus \{0\}$ and pairwise different monomials $m_d, \dots, m_0$ for some non-negative integer d (the empty sum is seen equivalent to 0). We write $R[x_1, \dots, x_n]$ for the set of all polynomials with coefficients from R and variables $x_1, \dots, x_n$. If a polynomial has only one variable, then it is called univariate, else it is called multivariate.
The degree of x_i in a monomial $x_1^{d_1} * \dots * x_n^{d_n}$ is its exponent d_i and the total degree of the monomial is the sum of the degrees of all variables appearing in that monomial. The degree of x_i in the polynomial p is its maximal degree under all monomials.
A monomial is linear if it has total degree less than or equal to one. Otherwise, it is nonlinear and similarly for polynomials.
Formulas without multiplication between variables, i.e. considering only linear polynomials, form the logic of linear real arithmetic (LRA).
Please note that in this thesis work we will be dealing with quantifier-free LRA and NRA. So, from now on if we use the words LRA and NRA, that means we are talking about quantifier-free LRA and NRA; sometimes we write QF_NRA and QF_LRA to emphasize this fact.

Example 2.3.1. *The following is a LRA formula:*

$$\varphi = (2x + 4y \leq 0) \wedge (6x + 4z < 0)$$

Here, x, y, z are monomials, $2x, 4y, 6x, 4z$ are terms. As it is a linear formula, each variable has a degree of 1 in φ.

Example 2.3.2. *The following is a NRA formula:*

$$\varphi = (\underbrace{x^5 + 2x + 4z}_{p_1} \leq 0) \wedge (\underbrace{6xy^3 + 4z}_{p_2} < 0)$$

Here, x^5, x, z, xy^3 are monomials, $2x, 4z, 6xy^3$ are terms. The degree of x in p_1 and p_2 is 5 and 1, respectively.

2.4 Satisfiability Modulo Theories Solving

Satisfiability Modulo Theories (SMT) solving [KS16, BHvMW09] checks the satisfiability of FO logic formulas with respect to a background theory [dMDS07]. It is called satisfiability because we assume quantifier-free formulas whose satisfiability should be decided and modulo theories because different theories can be considered.
The satisfiability checking problem for propositional logic is the problem of determining whether a given propositional logic formula is satisfiable. A propositional logic formula is said to be satisfiable if there exists a variable assignment that makes the formula *true*, otherwise unsatisfiable. One of the most successful technologies for this

test is SAT solving. A SAT solver solves the satisfiability checking problem by implementing a decision procedure. The Davis–Putnam–Logemann–Loveland (DPLL) algorithm is the basis for most modern SAT solvers [BHvMW09]. The DPLL algorithm expects the input formula to be in a conjunctive normal form (CNF) defined in the following.

Definition 2.4.1. *(Conjunctive Normal Form). A propositional logic formula is said to be in Conjunctive Normal Form (CNF) if and only if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal is either a positive or a negative instance of a boolean variable. A CNF formula φ is defined as follows:*

$$\varphi = \bigwedge_{i=1 \ldots n} C_i$$

$$C_i = \bigvee_{j=1 \ldots m_i} a_{ij}$$

Here, C_i is the i^{th} clause and n is total the number of clauses in φ, a_{ij} is a literal and m_i is the total number of literals in C_i.

Example 2.4.1. *The following is a CNF formula:*

$$\varphi = \underbrace{(x)}_{C_1} \wedge \underbrace{(\neg x)}_{C_2} \wedge \underbrace{(\neg x \vee y)}_{C_3} \wedge \underbrace{(\neg x \vee \neg y)}_{C_4}$$

where $x, \neg x, y$ and $\neg y$ are literals and $C_1, \ldots, C_4$ are clauses.

A theory solver for a theory T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T-satisfiable [Bar].
Figure 2.1 shows the main structure of an SMT solver. A SAT solver searches for a satisfying solution for the boolean skeleton of the problem. That means, it does not look into the theory; it checks the abstraction of the formula. In order to be complete, the SAT solver communicates with the theory solver which extends the logical search by checking theory constraints. The communication is in two directions. First, the SAT solver sends a request to the theory solver and the theory solver gives some feedback about the result to the SAT solver. If we have a certain logic which we want to solve, for example, LRA, we have a SAT solver to handle the boolean structure of the LRA formula and we also have e.g. a simplex solver [DdM06] as theory solver to check conjunctions of LRA constraints.

2.5 Approximation-Based Approach

NRA formulas are in general hard to solve and the applicability is restricted in practice because the decision procedures are computationally intensive. That is why we perform linearization for NRA formulas.
Our thesis work is inspired by the Ph.D. thesis work, "Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions" [Irf18]. The authors of [CGI+18] deal with the problems of SMT and Verification Modulo Theories (VMT), based on checking the satisfiability of formulas of quantifier-free NRA and quantifier-free NRA augmented with transcendental

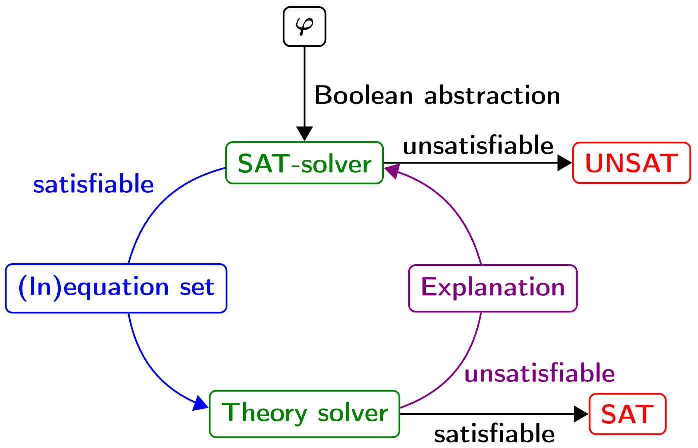


Figure 2.1: Full lazy SMT solver [Ábr]

functions (NTA). Transcendental functions include the exponential function, the logarithm and the trigonometric functions.

The main idea of [CGI$^+$18] is to abstract non-linear multiplication and transcendental functions as uninterpreted functions (UFs) as described in Section 2.5.1 below. This abstraction is performed iteratively over an abstract domain containing LRA and uninterpreted functions. The theory of uninterpreted functions (UFs) is the first-order theory with no restriction on the signature $\sum$, the set of non-logical symbols. In [CGI$^+$18], uninterpreted functions are used to model non-linear and transcendental functions that are iteratively and incrementally axiomatized with a lemma-on-demand approach [CGI$^+$18]. In other words, a refinement of the abstraction is performed and spurious interpretations are eliminated to be closer to satisfying solutions of the NRA formula. The authors named this abstraction-refinement approach as Incremental Linearization (IL).

Multiplication between variables is abstracted as a binary uninterpreted function. Detection of spurious models helps to tighten the abstraction. After detecting the spurious models, for each abstraction-refinement loop, some linear constraints are added to the input formula loop to tighten the abstraction. The linear constraints include tangent planes resulting from differential calculus and monotonicity constraints [CGI$^+$18]. On the other hand, each transcendental function is abstracted as a unary uninterpreted function. Taylor series is used to compute the coefficients. When spurious models are found, the piecewise-linear axioms are instantiated with upper and lower bound. For transcendental functions, the abstraction refinement is based on the addition.

In [CGI$^+$18], the authors have explained their contributions for the SMT case briefly and also described the extension of IL from the SMT to the VMT case. However, we are only concerned about the details of the refinement mechanisms and of the detection of satisfiable results for SMT case only. Also, this thesis work does not deal with transcendental functions. So, we will not be going into the details of the authors' contributions regarding the transcendental functions for SMT case.

Next we give a brief explanation of solving NRA formulas by IL for SMT case according to [CGI$^+$18].

2.5.1 Incremental Linearization for SMT (NRA)

In [CGI+18], the authors have proposed Algorithm 1 listed below, for checking the satisfiability of quantifier-free NRA formulas.
The input is an NRA formula φ. The algorithm returns a boolean value if φ is detected to be satisfiable or unsatisfiable. If φ is unsatisfiable, it also returns a set Γ of constraints from the combined theory of uninterpreted functions and linear real arithmetic (UFLRA). If φ is satisfiable, it returns the empty set $\emptyset$.

Algorithm 1 The main algorithm SMT-NRA-CHECK [CGI+18]

```
SMT-NRA-CHECK($\varphi$)
 1: $\varphi' :=$ SMT-PREPROCESS($\varphi$)
 2: $\hat{\varphi} :=$ SMT-INITIAL-ABSTRACTION($\varphi'$)
 3: $\Gamma := \emptyset$
 4: while true do
 5:     $\langle sat, \hat{\mu} \rangle :=$SMT-UFLRA-CHECK($\hat{\varphi} \wedge \bigwedge_{c \in \Gamma} c$)
 6:     if not sat:
 7:         return $\langle false, \Gamma \rangle$
 8:     $\langle sat, \Gamma' \rangle :=$CHECK-REFINE($\varphi, \hat{\varphi}, \hat{\mu}$)
 9:     if sat:
10:         return $\langle true, \emptyset \rangle$
11:     $\Gamma := \Gamma \cup \Gamma'$
```

In line 1, a method SMT-PREPROCESS is invoked which performs a preprocessing step. This preprocessing step generates a formula $\varphi' = \varphi \wedge \varphi_{shift}$ where φ_{shift} defines the values of some fresh real variables w_x in terms of some variables x in φ. Then in line 2, the method SMT-INITIAL-ABSTRACTION performs the abstraction of the formula φ' and returns an abstracted formula that is assigned to $\hat{\varphi}$. The abstraction means recursively replacing each non-linear term e.g., $x * y$ by an uninterpreted function application $f_*(x, y)$. So, the authors have only used the uninterpreted function for abstraction. No abstraction is performed for linear multiplications, e.g., $c*x$ where c is a constant, hence all the linear multiplications of φ' remain unchanged.
The set of constraints Γ is initialized to the empty set $\emptyset$. Lines 4 to 11 is a loop where at each iteration the abstraction is refined by extending the set of UFLRA constraints Γ that is responsible for removing spurious solutions. The satisfiability checking of the formula $\hat{\varphi} \wedge \bigwedge_{c \in \Gamma} c$ is performed by invoking the method SMT-UFLRA-CHECK with the formula $\hat{\varphi} \wedge \bigwedge_{c \in \Gamma} c$ as input (line 5). The method SMT-UFLRA-CHECK calls an SMT solver for UFLRA, the combined theory of linear arithmetic and uninterpreted functions. This method returns either *true* with the current satisfying abstract model $\hat{\mu}$ or *false* with an unsatisfiable core defined as follows:

Definition 2.5.1. *(Unsatisfiable Core). An unsatisfiable core of an unsatisfiable CNF formula is a subset of the clauses whose conjunction is unsatisfiable.*

The loop breaks if the formula $\hat{\varphi} \wedge \bigwedge_{c \in \Gamma} c$ is unsatisfiable, meaning that also the input NRA problem φ is unsatisfiable (lines 6 to 7). Otherwise, in line 8, the method CHECK-REFINE takes as input the formula φ, the abstracted formula $\hat{\varphi}$ and an abstract model $\hat{\mu}$ and returns either *true* with the empty set $\emptyset$ or *false* with a non-empty set of UFLRA constraints Γ'. A detailed explanation of the method CHECK-REFINE is given in the next Section 2.5.2. There is another loop-breaking condition from lines 9 to 10. The algorithm enters the lines 9 and 10 if CHECK-REFINE returns *true* with $\emptyset$ what means that the solution for the abstraction also satisfies the original input NRA formula φ. When none of the mentioned loop breaking conditions occur, the non-empty set of UFLRA constraints Γ' is added to Γ at line 11 and then the next iteration starts.

2.5.2 Abstraction Refinement and Spuriousness Check

The authors propose Algorithm 2 for checking the spuriousness of the abstract solution and refining the abstraction. Algorithm 2 shows that the method CHECK-REFINE takes the original NRA formula φ, the abstracted formula $\hat{\varphi}$ and the model $\hat{\mu}$ for the abstracted formula as inputs. In line 1, a method CHECK-NRA-MODEL is called on φ and $\hat{\mu}$ and this method checks if the formula φ is satisfied by the model $\hat{\mu}$. When CHECK-NRA-MODEL returns *true*, the algorithm enters to line 2 and it returns *true* with empty set $\emptyset$, what will terminate the whole satisfiability checking process. The method CHECK-NRA-MODEL is described in Section 2.5.2.2.

Algorithm 2 The algorithm CHECK-REFINE [CGI+18]

```
CHECK-REFINE($\varphi, \hat{\varphi}, \hat{\mu}$)
1: if CHECK-NRA-MODEL($\varphi, \hat{\mu}$)
2:     return $\langle true, \emptyset \rangle$
3: $\Gamma$ := BLOCK-SPURIOUS-PRODUCT-TERMS($\hat{\varphi}, \hat{\mu}$)
4: return $\langle false, \Gamma \rangle$
```

If the algorithm does not enter to line 2, it means the abstracted model $\hat{\mu}$ is spurious and the abstraction needs to be refined. A model is spurious means it violates some multiplications in the original NRA formula φ [CGI+18]. The method BLOCK-SPURIOUS-PRODUCT-TERMS is invoked at line 3 to refine the abstraction and exclude the abstract model $\hat{\mu}$ from further search. The method BLOCK-SPURIOUS-PRODUCT-TERMS takes as inputs $\hat{\varphi}$ and $\hat{\mu}$ and returns a set of UFLRA formulas as described in the next Section 2.5.2.1. This set of UFLRA formulas is stored in Γ and returned with the boolean result *false* in line 4 which influences the whole process to be continued further and to proceed for the next loop iteration (Algorithm 1, lines 9 to 10). So, when Algorithm 2 terminates, then either the original formula φ is found satisfiable, or a set of refinement constraints is being created.
This was a high-level description of the abstraction refinement and spuriousness check. We describe the concept of this refinement and spuriousness check-in details in the following Sections 2.5.2.1 and 2.5.2.2, respectively.

2.5.2.1 Abstraction Refinement for NRA

In [CGI+18], multiplications terms are refined in the function BLOCK-SPURIOUS-PRODUCT-TERMS. To perform the refinement, they have provided some constraint schemata, shown in Figure 2.1 which are tautologies in NRA. These constraint schemata prevent spurious assignments of the abstract model to multiplication terms. In the BLOCK-SPURIOUS-PRODUCT-TERMS method, it is checked whether the values of multiplication terms satisfy the constraint schemata. Then the method collects all the unsatisfied constraints from the constraints schemata into a list and returns it to the caller method CHECK-REFINE (Algorithm 2, line 3).

In Figure 2.1, we can see five types of refinement constraints namely Zero, Sign, Commutativity, Monotonicity and Tangent-plane where x, x_i, y, y_i are variables and a, b are rational values. It is straightforward to verify that all the constraints are valid formulas in any theory interpreting $f_*()$ as $*$ [CGI+18]. Each Zero constraint has a single multiplication term and that is why the refinements performed by Zero constraints are called single-term refinements. On the other hand, the refinements performed by Sign, Commutativity and Monotonicity constraints are called double-term refinements as they involve pairs of multiplication terms. Moreover, Tangent-plane constraints refer to a single multiplication term and a single point (a, b) and the refinements by these is also called single-term refinements.
Before explaining the above-mentioned different types of refinements in a formal way, it is needed to explain the Tangent-plane constraints as it is the interesting part of all refinement constraints, whereas the Zero, Sign, Commutativity and Monotonicity constraints are self-explanatory. Notice that, Tangent-plane constraints have equality constraints and inequality constraints. The equality constraints enforce the correct value of $f_*(x, y)$ at $x = a$ or $y = b$ and provide multiplication lines. On the other hand, the inequality constraints provide bound for $f_*(x, y)$ while x and y are not on multiplication lines. Figures 2.2a and 2.2b illustrate the surface of the uninterpreted funtion $f_*(x, y) = x * y$ and the top view of the surface, respectively. This kind of surface is known in geometry as hyperbolic paraboloid.

Definition 2.5.2. *(Hyperbolic Paraboloid). A hyperbolic paraboloid is a doubly-ruled surface, i.e., for every point on the surface, there are two distinct lines on the surface such that they pass through the point [CGI+18].*

Definition 2.5.3. *(Tangent Plane). A tangent plane at a point (a, b) to a bivariate function $f(x, y)$ is defined as:*

$$f(a, b) + \frac{d}{dx} f(a, b) * (x - a) + \frac{d}{dy} f(a, b) * (y - b).$$

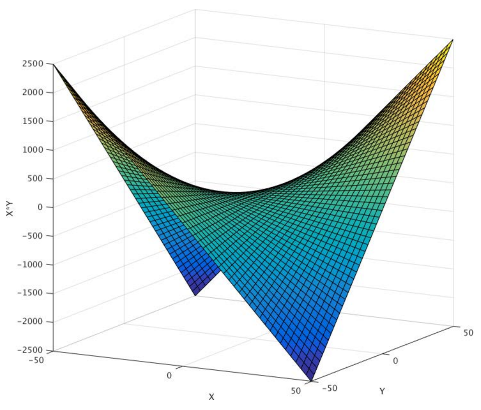
For multiplication, this yields the tangent plane $a * y + b * x - a * b$ [Har].

In Figure 2.2c we can see that at each point on a hyperbolic paraboloid there is a tangent plane to the hyperbolic paraboloid surface as defined in Definition 2.5.3. Interestingly, the two lines on the surface going through the point are also in the tangent plane. In other words, the two projected lines define how the plane cuts the surface.

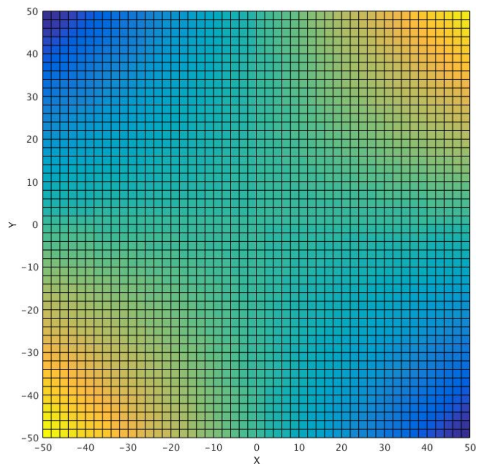
Different Types of Refinements:

- Let us take a constraint schema $\forall x, y.\psi$ from Figure 2.1, i.e., a Zero, Sign, Commutativity or Tangent-plane constraint schema. For each uninterpreted function application $f_*(t, s)$ in the abstraction $\hat{\varphi}$, let ψ' results from ψ by replacing x and y by t and s, respectively; if this formula evaluates to *false* under $\hat{\mu}$, then we add ψ' to an initially empty set $ST_*^{\hat{\mu}}$ of refinement formulas.

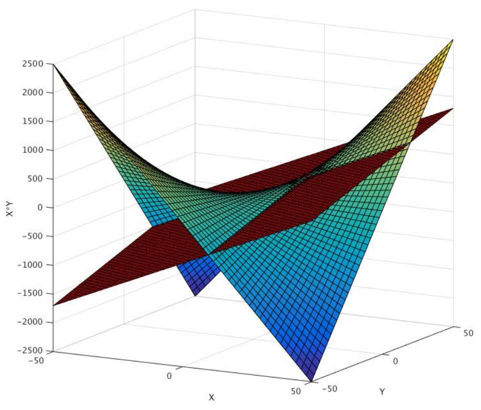
- Similarly, let us take a constraint schema $\forall x_1, y_1, x_2, y_2.\psi$ from Figure 2.1, i.e., Monotonicity constraint schema. For each pair of uninterpreted function applications $f_*(t_1, s_1)$ and $f_*(t_2, s_2)$, if the formula ψ' that results from ψ by substituting t_1, t_2, s_1, s_2 for x_1, x_2, y_1, y_2, respectively, then we add ψ' to $ST_*^{\hat{\mu}}$.



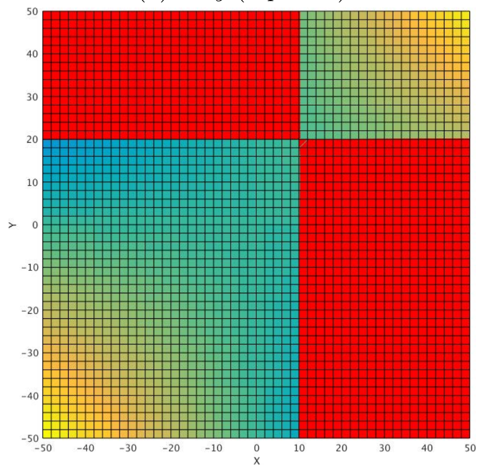
(a) $x * y$



(b) $x * y$ (top view)



(c) $x * y$ and tangent plane



(d) $x * y$ and tangent plane (top view)

Figure 2.2: Multiplication function and tangent plane (these pictures are taken from [CGI+18])

Example 2.5.1. *Assume that the input NRA formula φ has the multiplication terms $t_1 * s_1$ and $t_2 * s_2$. These multiplication terms are abstracted by $f_*(t_1, s_1)$ and $f_*(t_2, s_2)$, respectively and an abstracted formula $\hat{\varphi}$ is being created.*

Let, $\hat{\mu}$ be an abstract model containing the assignments:

$$\hat{\mu}[t_1] = 2, \hat{\mu}[s_1] = 3, \hat{\mu}[f_*(t_1, s_1)] = 7, \hat{\mu}[t_2] = 3, \hat{\mu}[s_2] = -4, \hat{\mu}[f_*(t_2, s_2)] = 5$$

Then $\hat{\mu}$ violates the third Zero constraint for $t_2 * s_2$:

$$((t_2 < 0 \wedge s_2 > 0) \vee (t_2 > 0 \wedge s_2 < 0)) \leftrightarrow f_*(t_2, s_2) < 0$$

The assignment $\hat{\mu}$ does not violate any Sign and Commutativity constraints, but violates all Monotonicity constraints:

1. $((abs(t_1) \leq abs(t_2)) \wedge (abs(s_1) \leq abs(s_2))) \rightarrow abs(f_*(t_1, s_1)) \leq abs(f_*(t_2, s_2))$
2. $((abs(t_1) < abs(t_2)) \wedge (abs(s_1) \leq abs(s_2)) \wedge (s_2 \neq 0)) \rightarrow (abs(f_*(t_1, s_1)) < abs(f_*(t_2, s_2)))$
3. $((abs(t_1) \leq abs(t_2)) \wedge (abs(s_1) < abs(s_2)) \wedge (t_2 \neq 0)) \rightarrow (abs(f_*(t_1, s_1)) < abs(f_*(t_2, s_2)))$

The assignment $\hat{\mu}$ also violates the Tangent-plane constraint in the points $(2, 3)$ and $(3, -4)$:

1. *at the point $(2, 3)$:*

   $f_*(2, s_1) = 2 * s_1 \quad \wedge$

   $f_*(t_1, 3) = 3 * t_1 \quad \wedge$

   $((t_1 > 2 \wedge s_1 < 3) \vee (t_1 < 2 \wedge s_1 > 3)) \rightarrow f_*(t_1, s_1) < 3 * t_1 + 2 * s_1 - 6 \quad \wedge$

   $((t_1 < 2 \wedge s_1 < 3) \vee (t_1 > 2 \wedge s_1 > 3)) \rightarrow f_*(t_1, s_1) > 3 * t_1 + 2 * s_1 - 6$

2. *at the point $(3, -4)$:*

   $f_*(3, s_2) = 3 * s_2 \quad \wedge$

   $f_*(t_2, -4) = -4 * t_2 \quad \wedge$

   $((t_2 > 3 \wedge s_2 < -4) \vee (t_2 < 3 \wedge s_2 > -4)) \rightarrow f_*(t_2, s_2) < -4 * t_2 + 3 * s_2 + 12 \quad \wedge$

   $((t_2 < 3 \wedge s_2 < -4) \vee (t_2 > 3 \wedge s_2 > -4)) \rightarrow f_*(t_2, s_2) > -4 * t_2 + 3 * s_2 + 12$

Finally, all these constraints are added to $ST_*^{\hat{\mu}}$ and returned to refine the abstracted formula $\hat{\varphi}$ which block the spurious model $\hat{\mu}$.

2.5.2.2 Spuriousness Check and Detecting Satisfiability

So far, we have seen how we can perform abstraction refinement by ruling out spurious models. Now we describe the behavior of the method CHECK-NRA-MODEL. CHECK-NRA-MODEL is a representation of the method CHECK-MODEL for NRA which detects the satisfiability of the formula φ and checks if the model $\hat{\mu}$ is spurious (Algorithm 3). The authors of [CGI+18] do not want that the algorithm only checks the spuriousness of $\hat{\mu}$, but also detect models and for that they want the algorithm to look for an actual model for φ "in the surroundings" of $\hat{\mu}$ [CGI+18], however this procedure of "model repair" is not implemented in this thesis and skipped in the following.
Line 1 extracts the NRA constraints in φ whose linearized abstractions hold under $\hat{\mu}$ [CGI+18]. Here, $T(\varphi, \hat{\mu})$ contains all atoms in φ whose abstraction is *true* under $\hat{\mu}$

Zero:

$\forall x, y.(x = 0 \vee y = 0) \leftrightarrow f_*(x, y) = 0$

$\forall x, y.((x > 0 \wedge y > 0) \vee (x < 0 \wedge y < 0)) \leftrightarrow f_*(x, y) > 0$

$\forall x, y.((x < 0 \wedge y > 0) \vee (x > 0 \wedge y < 0)) \leftrightarrow f_*(x, y) < 0$

Sign:

$\forall x, y.f_*(x, y) = f_*(-x, -y)$

$\forall x, y.f_*(x, y) = -f_*(-x, y)$

$\forall x, y.f_*(x, y) = -f_*(x, -y)$

Commutativity:

$\forall x, y.f_*(x, y) = f_*(y, x)$

Monotonicity:

$\forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2)) \wedge (abs(y_1) \leq abs(y_2))) \rightarrow$
$(abs(f_*(x_1, y_1)) \leq abs(f_*(x_2, y_2)))$

$\forall x_1, y_1, x_2, y_2.((abs(x_1) < abs(x_2)) \wedge (abs(y_1) \leq abs(y_2)) \wedge (y_2 \neq 0)) \rightarrow$
$(abs(f_*(x_1, y_1)) < abs(f_*(x_2, y_2)))$

$\forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2)) \wedge (abs(y_1) < abs(y_2)) \wedge (x_2 \neq 0)) \rightarrow$
$(abs(f_*(x_1, y_1)) < abs(f_*(x_2, y_2)))$

Tangent plane:

$\forall x, y.(f_*(a, y) = a * y) \wedge (f_*(x, b) = x * b) \wedge$
$((x > a \wedge y < b) \vee (x < a \wedge y > b)) \rightarrow f_*(x, y) < b * x + a * y - a * b \wedge$
$((x < a \wedge y < b) \vee (x > a \wedge y > b)) \rightarrow f_*(x, y) > b * x + a * y - a * b$

Table 2.1: The refinement UFLRA constraint schemata for multiplication, where a, b are constants and x, y are variables [CGI$^+$18]

Algorithm 3 The algorithm CHECK-NRA-MODEL [CGI+18]

```
CHECK-NRA-MODEL $(\varphi, \hat{\mu})$
1: $\psi := \bigwedge_{A \in T(\varphi,\hat{\mu})} A \ \wedge \bigwedge_{B \in F(\varphi,\hat{\mu})} \neg B$
2: return $\hat{\mu} \models \psi$
```

and $F(\varphi, \hat{\mu})$ all those that are *false* under $\hat{\mu}$. The algorithm returns whether the NRA concretizations of the abstract constraints have the same truth value, i.e., whether $\hat{\mu}$ satisfies also φ.
The idea of this algorithm is demonstrated by the following example [CGI+18]:

Example 2.5.2. *Consider the following NRA formula φ and the abstracted formula $\hat{\varphi}$:*

$$\varphi := (x * y = 10) \wedge (2 \leq x \leq 4) \wedge (2 \leq y \leq 4)$$

$$\hat{\varphi} := (f_*(x, y) = 10) \wedge (2 \leq x \leq 4) \wedge (2 \leq y \leq 4)$$

Assume, SMT-UFLRA-CHECK returns a model $\hat{\mu}$ (Algorithm 1, line 5):

$$\hat{\mu}[x] = 2, \quad \hat{\mu}[y] = 4, \quad \hat{\mu}[f_*(x, y)] = 10$$

So, the method CHECK-REFINE is invoked (Algorithm 1, line 8) and there is still a chance to find a model for the original formula φ by using a method CHECK-NRA-MODEL (Algorithm 2, line 1 to 2). But $2 * 4 \neq 10$ and this inequality makes $\hat{\mu}$ spurious and CHECK-NRA-MODEL returns *false*.

Chapter 3

Incremental Linearization for Real Arithmetic

3.1 System Architecture

So far, we have seen the basic concepts to understand our thesis work and how the authors of [CGI$^+$18] have performed incremental linearization (IL) using SMT solving. Figure 3.1 depicts the whole process of our thesis work.
The input is an NRA formula φ. At first, φ is linearized to a LRA formula $\hat{\varphi}$ by a linearization method. This linearization method follows Algorithm 5 which we will explain later. The formula $\hat{\varphi}$ is passed to the SMT solver and it results in either SAT with a satisfying solution $\hat{\mu}$ of $\hat{\varphi}$ or UNSAT or UNKNOWN. SAT means the SMT solver finds $\hat{\varphi}$ satisfiable, UNSAT means $\hat{\varphi}$ is not satisfiable and UNKNOWN means the SMT solver is unable to determine the satisfiability of $\hat{\varphi}$. If the SMT solver results in either UNSAT or UNKNOWN, the method terminates. Otherwise, it continues to work following the next step.
The next step is to extend $\hat{\mu}$ to a model μ for assigning all variables in φ. We have named μ as an estimated model. If μ satisfies φ, it returns SAT and we are done. Otherwise, we have to perform refinement over some predefined axioms to resist the spurious solution μ. This refinement process outputs a set A of axioms that prevents spurious solutions including μ. Finally, $\hat{\varphi}$ is augmented by adding each axiom ψ from the set A and again $\hat{\varphi}$ is passed to the SMT solver. This loop continues until the SMT solver finds UNSAT or UNKNOWN for $\hat{\varphi}$ or the estimated model μ satisfies φ.

3.2 Our Approach

In our implementation we will use SMT-RAT [CKJ$^+$15], an open source C++ toolbox for strategic and parallel SMT solving. SMT-RAT offers a collection of modules, under others for solving quantifier-free linear as well as non-linear real and integer arithmetic formulas.
SMT-RAT modules implement a common interface. The interface consists of some methods including addCore and checkCore. The addCore(φ_i) method is invoked to add formulas that should be satisfied, i.e., the conjunction $\varphi = \wedge_{i=1}^{k} \varphi_i$ of all added

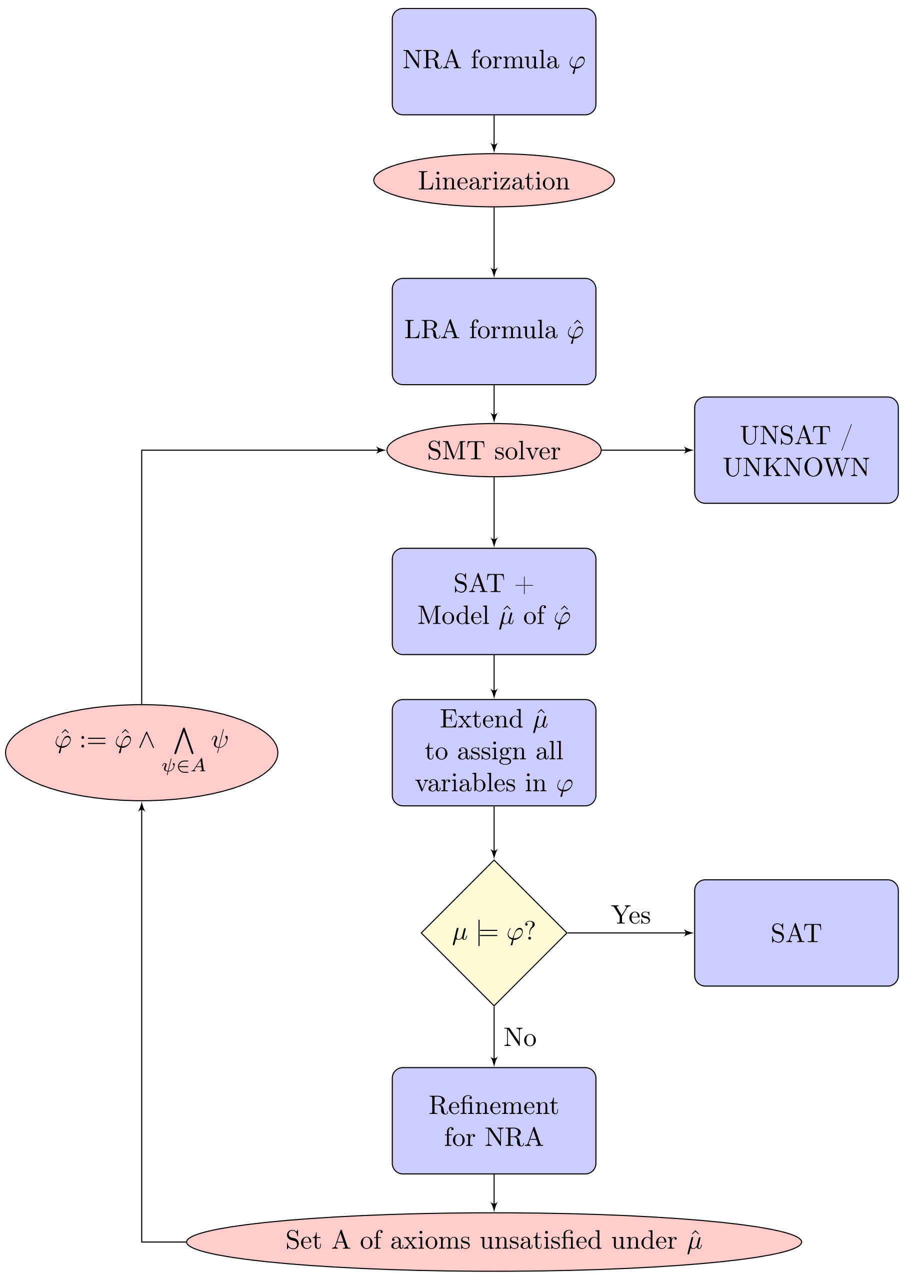

NRA formula φ
Linearization
LRA formula $\hat{\varphi}$
SMT solver
UNSAT / UNKNOWN
SAT + Model $\hat{\mu}$ of $\hat{\varphi}$
Extend $\hat{\mu}$ to assign all variables in φ
$\mu \models \varphi$?
Yes
SAT
No
Refinement for NRA
Set A of axioms unsatisfied under $\hat{\mu}$
$\hat{\varphi} := \hat{\varphi} \wedge \bigwedge_{\psi \in A} \psi$

Figure 3.1: The incremental linearization process

formulas will be the subject of the satisfiability check. The checkCore() method is invoked at the end to check the satisfiability of φ.
We have constructed an SMT-RAT module named *NRAILModule* that linearizes an input NRA formula incrementally, solves it and utilizes the solution if the input NRA formula is not satisfied by it to make it satisfying via a refinement process. In our case, we perform linearization for each φ_i in addCore(φ_i) before their addition. The satisfiability checking logic with refinement process is formulated in checkCore().

Example 3.2.1. *Assume an example quantifier-free NRA formula:*

$$\varphi := (3x^9y^2 - y < 5) \wedge (xy + 1 < 0)$$

SMT-RAT will reorganize this fomula as follows, we will use this example formula for other examples too:

$$\varphi := (3x^9y^2 + (-1)y + (-5) < 0) \wedge (xy + 1 < 0)$$

Here, the set of sub-formulas is, $\{3x^9y^2 + (-1)y + (-5) < 0, xy + 1 < 0\}$*. Each sub-formula from this set is denoted as* φ'*.*

To understand this paper, we do not need to know the internal methodology of SMT-RAT in depth. We will explain SMT-RAT partly whenever we will need it to understand our work.

3.2.1 addCore(φ_i)

The algorithm addCore(φ_i) is shown in Algorithm 4. This addCore(φ_i) is responsible for invoking the linearization method for each φ_i. SMT-RAT has no module or method to linearize a formula. That is why we have introduced a method for linearization. Also, SMT-RAT does not support uninterpreted functions. So, in this work, one of our contributions is to adapt the linearization process by variables.
Whenever SMT-RAT runs *NRAILModule*, at first, it calls addCore(φ_i) for each input formula φ_i. Each φ_i is linearized by linearization(φ_i) and stored in $\hat{\varphi}_i$ (line 1). Finally, addSubformulaToPassedFormula($\hat{\varphi}_i$) is invoked (line 2) which adds $\hat{\varphi}_i$ to the set of (linearized) input formulas. The result, which is *false* if a conflict (unsatisfiability) is detected already during addition and *true* otherwise, is returned to the caller.

Algorithm 4 The algorithm addCore

```
addCore($\varphi_i$)
1: $\hat{\varphi}_i$ := linearization($\varphi_i$)
2: result := addSubformulaToPassedFormula($\hat{\varphi}_i$)
3: return result
```

3.2.1.1 Linearization

The basic idea is to abstract each multiplication term (i.e., $x * x$, $x * y$ and so on) in a formula by a new variable called z-variable (i.e., $z_1, z_2, \ldots, z_i$ for some $i \in \mathbb{N}$). The abstraction is performed incrementally until the formula is linearized. The algorithm for linearization is shown in Algorithm 5.

We store the input formula φ_i in $\hat{\varphi}_i$ (line 1) and linearize it for each non-linear constraint c in φ_i as follows. We extract the left-hand side of c which is basically a polynomial p (line 3). We build linear abstractions of all terms in p and store them in an initially empty list $\hat{p}$ (lines 5-13). For each term t, it is checked whether it is constant or linear, or non-linear. If t is constant or linear (line 6), then it is added as a linear term to the list $\hat{p}$ (line 7). If t is non-linear, then the else block applies (lines 8 to 12).

Algorithm 5 The algorithm linearization

```
linearization($\varphi_i$)
 1: $\hat{\varphi}_i$ := empty list
 2: for each non-linear constraint $c \in \varphi_i$
 3:     $p$ := left hand side of $c$
 4:     $\hat{p}$ := empty list
 5:     for each term $t \in p$:
 6:         if $t$ is a constant || $t$ is linear:
 7:             $\hat{p}$.pushBack($t$)
 8:         else
 9:             $m$ := monomial of $t$
10:             $a$ := coefficient of $t$
11:             $z$ := abstractMonomial($m$)
12:             $\hat{p}$.pushBack($a * z$)
13:     end
14:     replace $c$ by createFormula($\hat{p}$, relational operator in $c$) in $\hat{\varphi}_i$
15: end
16: return $\hat{\varphi}_i$
```

In lines 9 and 10, we extract the monomial m and the coefficient a from $t := a * m$, respectively. After that, abstractMonomial(m) is invoked which returns a variable z that is used as an abstraction for m (line 11). The method abstractMonomial(m) is explained in Section 3.2.1.2. The variable z is achieved by replacing each univariate multiplication term of m recursively as follows:

$$\underbrace{\underbrace{\underbrace{\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}}_{z_2}*\underbrace{\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}}_{z_2}}_{z_3}*x}_{z\,\rightarrow\,z_4}$$

In multivariate monomials, first all univariate terms x^d are abstracted, resulting in products $z_1 * \cdots * z_i$, in which we iteratively abstract each multiplication by another fresh variable from left to right (see Equation 3.1, Example 3.2.2). In line 12 the linear abstraction $a * z$ is added to the list $\hat{p}$ and the loop continues if there are other terms left in p.
After termination of the inner loop, in line 14, a linear constraint is created by building a sum of the terms in $\hat{p}$ and comparing it to 0 according to the comparison operator of c. The linearization replaces each occurrence of c by this linear constraint. Note that due to the reorganization of input formulas by SMT-RAT, as shown in Example

3.2.1, the left-hand side of c is always a sum of terms, the latter being products of a constant and variables and the right-hand side is zero. After this linearization step, the outer loop continues if there are further non-linear constraints left in φ_i. Lastly, in line 16, the algorithm returns the linear formula $\hat{\varphi}_i$.

Example 3.2.2. *Consider the same example formula as in the Example 3.2.1:*

$$\varphi := (3x^9y^2 + (-1)y + (-5) < 0) \wedge (xy + 1 < 0)$$

At first, linearization(φ_1) is invoked for $\varphi_1 := (3x^9y^2 + (-1)y + (-5) < 0)$ (line 1, Algorithm 4).
Now,
constraint $c := (3x^9y^2 + (-1)y + (-5) < 0)$ (line 2, Algorithm 5)
left hand side $p := 3x^9y^2 + (-1)y + (-5)$ (line 3, Algorithm 5)
terms are: $3x^9y^2, (-1)y$ and -5.
Now, for each term t the for loop will be run (line 5 to 13, Algorithm 5):

- *For $t := 3x^9y^2$ (line 9 to 12):*

  $$\text{monomial } m := x^9y^2 \text{ and coefficient } a := 3$$

  This monomial is linearized as follows according to Algorithm 6:

  $$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}}_{z_2}*\underbrace{\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}}_{z_2}}_{z_3}*x}_{z_4}*\underbrace{y*y}_{z_5}}_{z_6} \tag{3.1}$$

  $$\text{So, } z := z_6 \text{ and } \hat{p}[0] := 3 * z_6$$

- *For $t := (-1)y$, t is already linear (line 7):*

  $$\hat{p}[1] := (-1)y$$

- *For constant term $t := -5$ (line 7):*

  $$\hat{p}[2] := -5$$

  So,

  $$\text{linearized polynomial} \quad \hat{p} := z_6 + (-1)y + (-5)$$

  $$\text{linearized subformula} \quad \hat{\varphi}_1 := (z_6 + (-1)y + (-5) < 0)$$

  Then $\hat{\varphi}_1$ is returned to addCore(φ_1) and $\hat{\varphi}_1$ is inserted to $\hat{\varphi}$ by addSubformulaToPassedFormula ($\hat{\varphi}_1$) (line 2, Algorithm 4). In the same way, linearization(φ_2) is invoked for $\varphi_2 := x * y + 1 < 0$ (line 1, Algorithm 4) and we get linear subformula $(z_7 + 1 < 0)$ as follows:

  $$(\underbrace{x*y}_{z_7} + 1 < 0)$$

  Finally, we get the linearized formula for φ:

  $$\hat{\varphi} := (z_6 + (-1)y + (-5) < 0) \wedge (z_7 + 1 < 0)$$

3.2.1.2 Abstract a Monomial Incrementally by z-variables

In the previous section, we have seen how the linearization process works and when abstractMonomial(m) is called. We have also seen how the abstractMonomial(m) method is executed and what this method returns (see Equation 3.1, Example 3.2.2). Encapsulating m incrementally by z-variables is the core of the whole linearization process. In this section, we are going to see the algorithm behind this abstraction of m.

Algorithm 6 The algorithm abstractMonomial

```
abstractMonomial($m$)
1: $vList$ := empty list
2: for each variable $v$ with $v^d \in m$ where $d$ is a positive integer:
3:     $vList$.pushBack(abstractUnivariateMonomial($v, d$))
4: end
5: $z$ := abstractProductRecursively($vList$)
6: return $z$
```

Algorithm 7 The algorithm abstractProductRecursively

```
abstractProductRecursively($vList$)
1: while ($vList$.size() > 1) do
2:     $first$ := $vList$.popFront()
3:     $second$ := $vList$.popFront()
4:     $z$ := getAbstractVariable($first * second$)
5:     $vList$.pushFront($z$)
6: return $vList$.front()
```

The main idea of Algorithm 6 is to replace univariate multiplication terms v^d in the input monomial m, where $d > 1$, by fresh z-variables; the abstracting z-variable remains the same for all occurrences of v^d. This replacement continues until in the input monomial $m = x_1^{d_1}, \ldots, x_n^{d_n}$ with $d_i \geq 1$ for all $i = 1, \ldots, n$, we introduced an abstraction variable z_i for each $x_i^{d_i}$ with $d_i > 1$. Finally, the product of these abstraction variables and all x_i with $d_i = 1$ is abstracted by Algorithm 7 and the result is returned to the caller method linearization(φ_i):

$$\underbrace{\underbrace{\underbrace{vList_0 \quad * \quad vList_1}_{z_j} \quad * \quad vList_2}_{\substack{z_{j+1} \\ \vdots}} \quad * \quad \ldots \quad * \quad vList_i}_{z \,\to\, z_{j+i-1}} \tag{3.2}$$

Here, i is a non-negative integer and j is a positive integer.

3.2.1.3 Abstraction of Univariate Monomials

From Section 3.2.1.2 we have got to know that each v^d, $d > 1$, in a monomial m is abstracted by a fresh variable (line 3, Algorithm 6). The abstraction uses abstractUnivariateMonomial(v, d) in Algorithm 8. We assume the exponent d to be positive.

Algorithm 8 The algorithm abstractUnivariateMonomial

```
abstractUnivariateMonomial($v, d$)
1: extraVList := empty list
2: while ($d > 1$):
3:     if $d\%2 == 1$:
4:         $d := d - 1$
5:         extraVList.pushFront($v$)
6:     $d := d/2$
7:     $v :=$ getAbstractVariable($v^2$)
8: extraVList.pushFront($v$)
9: return abstractProductRecursively(extraVList)
```

As long as the current univariate monomial is not linear, i.e. as long as $d > 1$, we iterate (line 2) as follows. If d is odd (line 3), then the exponent is decreased by one (line 4) and it is remembered in an initially empty list (line 1) that the abstraction result needs to be multiplied by v (line 5). Now d is even, so each pair of v will be easily replaced by the same z-variable (lines $6-7$). This procedure is repeated until we get a single variable, i.e., until $d = 1$. We push the variable into the list extraVList (line 8) and call abstractProductRecursively(extraVList) to compute an abstraction for the product of all variables in the list which is returned (line 9).

Example 3.2.3. *Let us consider a variable x with an odd exponent, $d = 7$. So, abstractUnivariateMonomial$(x, 7)$ executes as follows:*
The exponent 7 is odd, therefore it is decreased by 1 and x is pushed into extraVList:

$$\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}*\underbrace{x*x}_{z_1}*x \longrightarrow \overset{\text{extraVList}}{\boxed{}}$$

$$d := \frac{7-1}{2} = 3 \qquad v := z_1$$

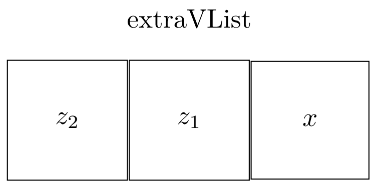
The exponent 3 is odd, therefore z_1 is pushed to extraVList:

$$\underbrace{z_1*z_1}_{z_2}*z_1 \longrightarrow \overset{\text{extraVList}}{\boxed{x}}$$

$$d := \frac{3-1}{2} = 1 \qquad v := z_2$$

Now the exponent is 1*,* z_2 *is pushed into the front of extraVList and the loop terminates. To get the single and final z-variable, the product of each two elements of extraVList are replaced as follows:*

$$\underbrace{\underbrace{z_2 * z_1}_{z_3} * x}_{z_4}$$

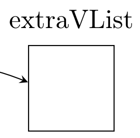


$$m := z_4$$

Example 3.2.4. *In Example 3.2.2 we have seen how a monomial* x^9y^2 *is linearized in an abstract way (Equation 3.1). In this example we will see how Algorithms 6 and 8 are executed to linearize the monomial* x^9y^2*. When abstractMonomial(*x^9y^2*) is invoked it executes as follows:*

- *For* x^9 *in* m*, abstractUnivariateMonomial(*$x, 9$*) is invoked (line* 3*, Algorithm 6). The exponent* 9 *is odd, therefore it is decreased by* 1 *and* x *is pushed into extraVList:*

$$\underbrace{x * x}_{z_1} * \underbrace{x * x}_{z_1} * \underbrace{x * x}_{z_1} * \underbrace{x * x}_{z_1} * x \longrightarrow \boxed{}$$



$$d := \frac{9-1}{2} = 4 \qquad v := z_1$$

Now the exponent $d = 4$ *is even:*

$$\underbrace{z_1 * z_1}_{z_2} * \underbrace{z_1 * z_1}_{z_2}$$

$$d := \frac{4}{2} = 2 \qquad v := z_2$$

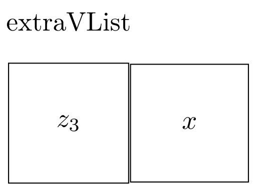
The exponent $d = 2$ *is still even:*

$$\underbrace{z_2 * z_2}_{z_3}$$

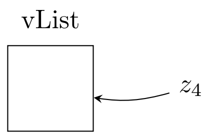
$$d := \frac{2}{2} = 1 \qquad v := z_3$$

Now, $d = 1$ *and the loop terminates. Then* z_3 *is pushed into the front of extraVList and the product of each two elements of extraVList are replaced as follows:*

$$\underbrace{z_3 * x}_{z_4}$$



The method abstractUnivariateMonomial($x, 9$) returns z_4 (line 9, Algorithm 8). Now, z_4 is pushed back into vList (line 3, Algorithm 6).

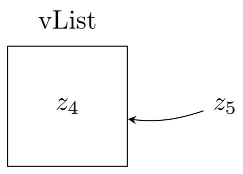


- *For y^2 in m, abstractUnivariateMonomial($y, 2$) is invoked (line 3, Algorithm 6). The exponent 2 is even:*

$$\underbrace{y * y}_{z_5}$$

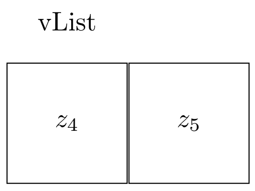
$$d := \frac{2}{2} = 1 \qquad v := z_5$$

So, abstractUnivariateMonomial($y, 2$) returns z_5 that is pushed back into vList (line 3, Algorithm 6).



- *Lastly, the product of each two elements of vList are replaced by z-variables from left to right according to Equation 3.2 and abstractMonomial(x^9y^2) returns z_6 (line 5 and 6, respectively, Algorithm 6).*

$$\underbrace{z_4 * z_5}_{z_6}$$



3.2.2 checkCore()

So far, we have seen the linearization process of the input formula φ done by addCore(φ_i) for each sub-formula φ_i of φ in detail. Now, further computations will be handled by checkCore(). This method will check the satisfiability of the linearized

formula $\hat{\varphi}$. If $\hat{\varphi}$ is satisfied, we will check whether the solution for the linear abstraction is also a solution for the non-linear formula φ. If it is the case, we are done. Otherwise, we will try to modify the solution for the linear abstraction in order to make it satisfying for φ. If we succeed, then we are done and φ is satisfiable. Otherwise, we refine the abstraction by adding some additional information about the properties of multiplication, expressed by linear formulas. This refinement process is the core functionality of checkCore() and our thesis work as well.
The algorithm checkCore is shown in Algorithm 9. A set A is initialized as empty (line 1). Then a while loop iterates until satisfiability can be decided (lines 2 to 11); note that, to assure termination, we could also restrict the number of loop iterations by adding an upper bound on it in the while-loop condition. In each loop iteration, the abstraction $\hat{\varphi}$ is passed to an SMT solver for LRA which returns either SAT or UNSAT or UNKNOWN (line 3). If it returns SAT, it will also return a satisfying model $\hat{\mu}$ for $\hat{\varphi}$ and our work starts. Otherwise, checkCore() terminates by returning UNSAT or UNKNOWN (lines 4 to 5).
Now, createEstimatedAssignment($\varphi, \hat{\varphi}, \hat{\mu}$) is invoked which will try to generate a model μ for φ by adapting the model $\hat{\mu}$ for $\hat{\varphi}$ (line 6). We have already mentioned in Section 3.2.1 that the linearization introduced in [CGI$^+$18] is modified not to use uninterpreted functions but variables for the abstraction. The adaptation is relatively straightforward, but the main difference is that now the solution to the linearization problem does not necessarily provide assignments to all variables. That is why we need some additional heuristic to get the assignments of variables which occur in the original formula φ. So, our idea is first to check if $\hat{\mu}$ contains any solutions for original variables and add it to μ. Then we guess values for the remaining original variables. In our work, we always guess the value zero. We have named μ estimated model. Once we find this extended model, we check if it satisfies φ (line 7). If φ is satisfied by μ, we get SAT (line 8 to 9). SAT means our assumption (if any) is correct. Otherwise, the refinement process is performed.

Algorithm 9 The algorithm checkCore

```
checkCore()
 1: $A := \emptyset$
 2: **while** *true* **do**
 3:     $\langle result, \hat{\mu} \rangle$ := SMT-LRA-Solver($\hat{\varphi}$)
 4:     **if** *result* is not sat:
 5:         **return** *result*
 6:     $\mu$ := createEstimatedAssignment($\varphi, \hat{\varphi}, \hat{\mu}$)
 7:     $\langle sat \rangle$ := isNRASatisfied($\varphi, \mu$)
 8:     **if** sat:
 9:         **return** sat
10:     $A$ := Refinement($\hat{\mu}$)
11:     $\hat{\varphi} := \hat{\varphi} \wedge \bigwedge_{\psi \in A} \psi$
```

Refinement is a process by which we try to remove the spuriousness of a linearized model $\hat{\mu}$ by adding some axioms to $\hat{\varphi}$ from a list of axioms so that the solution scopes get limited. We have already mentioned that our work is inspired by the work [CGI$^+$18]. We have used the same list of axioms shown in Figure 2.1 with some modifications. We will discuss our contribution to the refinement process in the next section. Refinement is responsible for extending the abstraction with axioms that are not satisfied by $\hat{\mu}$ (line 10). These unsatisfied axioms are collected into A. Lastly, each axiom $\psi \in A$ is conjoined to $\hat{\varphi}$ (line 11) and proceed to the next loop iteration with the refined $\hat{\varphi}$.

3.2.3 Refinement Process

The refinement process aims to prevent spurious assignments of the linearized model to multiplication terms, so that we can get closer and closer to the input NRA formula being satisfied. For each multiplication term, it is checked whether the linearized model satisfies all axioms shown in Figure 3.1. We are not concerned about satisfied axioms, but the unsatisfied axioms. Unsatisfied axioms are collected to a set A for each multiplication term (Algorithm 9, line 12). Each unsatisfied axiom of A is added to the linearized formula and again it is passed to the SMT LRA solver. Further computation remains the same as explained in Section 3.2.2.
The generation of axioms shown in Figure 2.1 is already done in [CGI$^+$18]. We generate those differently as we do not encode them as the uninterpreted function, but another way. So, our contribution is now to generate those in our way. That is why our list of axioms shown in Figure 3.1 is different from the list in Figure 2.1.

We have generated the axioms by encoding them as variables for each multiplication term. If we notice, we can see some other differences between the Figures 3.1 and 2.1:

- We have ignored the axioms of type Sign and Commutativity because we do not have parameters. That is why we do not have these expressions (i.e., $f(x, y), f(-x, -y), f(y, x)$ and so on) in the formula. We only have z and if we refine, then we will have x and y.

- We handle square expressions differently than normal multiplication by introducing a special axiom for squares instead of a tangent plane which is more appropriate. The axiom is similar. Instead of two values for the variables, we have only one value because the variables are the same. Let z_1^2 be abstracted by z means that $z = z_1 * z_1$ and their assignments are, $\hat{\mu}(z) = v$ and $\hat{\mu}(z_1) = v_1$. Now, if $v \neq v_1^2$, we will add our newly introduced axiom for square expression to A instead of the old one.

- Congruence for equalities is added as an axiom whether it satisfies for double multiplication terms. Let two multiplication terms be encoded as $z = z_1 * z_2$ and $z' = z_1' * z_2'$. The assignments are, $\hat{\mu}(z) = v, \hat{\mu}(z_1) = v_1, \hat{\mu}(z_2) = v_2, \hat{\mu}(z') = v', \hat{\mu}(z_1') = v_1'$ and $\hat{\mu}(z_2') = v_2'$.

$$\text{if } \neg((v_1 = v_1' \land v_2 = v_2') \rightarrow v = v'),$$

$$\text{add } ((z_1 = z_1' \land z_2 = z_2') \rightarrow z = z') \text{ to } A.$$

ICP Axioms: We have also contributed to this work by proposing an entirely new type of axioms based on integral constraint propagation (ICP). We named this type

Zero:

$\forall x, y.(x = 0 \lor y = 0) \leftrightarrow z = 0$

$\forall x, y.((x > 0 \land y > 0) \lor (x < 0 \land y < 0)) \leftrightarrow z > 0$

$\forall x, y.((x < 0 \land y > 0) \lor (x > 0 \land y < 0)) \leftrightarrow z < 0$

Monotonicity:

$$\forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2)) \land (abs(y_1) \leq abs(y_2))) \rightarrow$$
$$(abs(z_1) \leq abs(z_2))$$

$$\forall x_1, y_1, x_2, y_2.((abs(x_1) < abs(x_2)) \land (abs(y_1) \leq abs(y_2)) \land (y_2 \neq 0)) \rightarrow$$
$$(abs(z_1) < abs(z_2))$$

$$\forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2)) \land (abs(y_1) < abs(y_2)) \land (x_2 \neq 0)) \rightarrow$$
$$(abs(z_1) < abs(z_2))$$

Tangent plane:

$$\forall x, y.(z = a * y) \land (z = x * b) \land$$
$$(((x > a \land y < b) \lor (x < a \land y > b)) \rightarrow z < b * x + a * y - a * b) \land$$
$$(((x < a \land y < b) \lor (x > a \land y > b)) \rightarrow z > b * x + a * y - a * b)$$

$\forall x, y.(x = a \rightarrow z = a * x) \land (x \neq a \rightarrow z > 2 * a * x - a^2)$ (for square expr. $z = x^2$)

Congruence:

$\forall x_1, y_1, x_2, y_2.((x_1 = x_2) \land (y_1 = y_2)) \rightarrow (z_1 = z_2)$

Table 3.1: The list of axioms for refining multiplications $z = x * y$ and $z_i = x_i * y_i$, $i = 1, 2$

of axioms ICP axioms. Let us consider $z = x * y$ such that $\mu(z) \neq \mu(x) * \mu(y)$ and let us take the absolute values of x, y and z:

$$abs(\mu(x)) = a, \qquad abs(\mu(y)) = b \qquad abs(\mu(z)) = c$$

If one of a, b or c is zero or if $c = a * b$, then we apply the Zero axioms. Assume in the following $a, b, c \neq 0$ and $c \neq a * b$.
Now, if c is lower than $a * b$ (i.e., $c < a * b$), then we can decrease a and b such that

their product is still above c. We choose a' and b' in a way that $c' = a' * b'$ and $c < c' < a * b$ in order to exclude c. For example, we fix the value of a and modify the value of b by making it smaller. In Figure 3.2, we make b smaller by choosing any b' such that it is still larger than $\frac{c}{a}$ which is the first case in the following. Furthermore, it will still lead to a product $a * b'$ that is larger than c, but closer than c to $a * b$. Similarly, we can do it for a which is the second case in the following (Figure 3.3).

- $c < a * b'$ or, $\frac{c}{a} < b'$ and $0 < b' \leq b$
  We choose, $\frac{c}{a} < b' \leq b$
- $c < a' * b'$ or, $\frac{c}{b'} < a'$ and $0 < a' \leq a$
  We choose, $\frac{c}{b'} < a' \leq a$

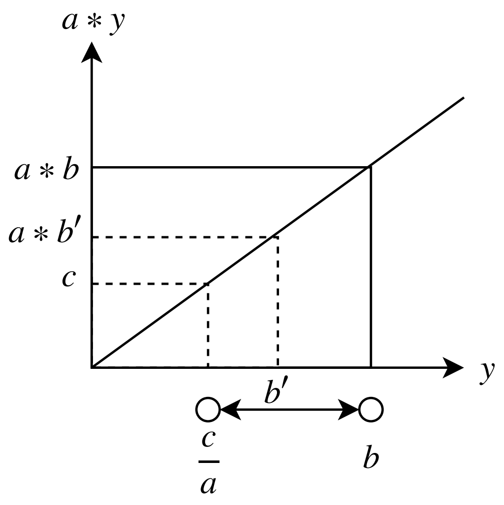


Figure 3.2: Fix a and modify b if $c < a * b'$

Notice that a and b are positive and we know that their product is not c.
So, for $c < a * b$ we have two axioms, generalizing the above observations also to the negative domain (remember we defined $c' = a' * b'$):

1. $$((x \geq a' \wedge y \geq b') \vee (x \leq -a' \wedge y \leq -b')) \rightarrow (z \geq c')$$

2. $$((x \geq a' \wedge y \leq -b') \vee (x \leq -a' \wedge y \geq b')) \rightarrow (z \leq -c')$$

Let us consider now $c > a * b$. That means we guessed the product too high. Dually to the previous case we take some values a' and b' above a and b respectively such that $a' * b'$ is still below c and state that for all pairs of values below a' and b' respectively, their product is less than $a' * b'$. This way, we say that $a * b$ is at most $a' * b'$. So, c will be excluded.
In the same way, we can choose a' and b' either the minimum (a and b, respectively) or arbitary near the maximum ($c + \epsilon$) or something in between:

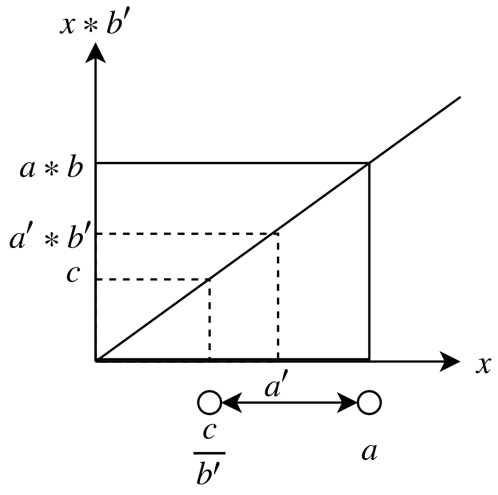


Figure 3.3: Fix b and modify a if $c < a' * b'$

- $c > a * b'$ or, $b' < \frac{c}{a}$ and $0 < b \leq b'$
  We choose, $b \leq b' < \frac{c}{a}$
- $c > a' * b'$ or, $a' < \frac{c}{b'}$ and $0 < a \leq a'$
  We choose, $a \leq a' < \frac{c}{b'}$

So, for $c > a * b$ we have one axiom, which covers also the case for zero values and the symmetric case for negative values:

$$(-a' \leq x \leq a') \land (-b' \leq y \leq b') \rightarrow (-c' \leq z \leq c')$$

Heuristics: We have also contributed to this work by using different heuristics to identify the unsatisfied axioms. Our primary goal of using different heuristics in the refinement process is to identify unsatisfied axioms in a way so that the axiom generation is computationally not too expensive and adding them for refinement will exclude a relatively large or search-wise "relevant" part of the state space. In other words, different heuristics will help us to find the best way of refinement. Generally, there are maximum number of unsatisfied axioms we can use for refinement. We may control the number of unsatisfied axioms to be added to A. We can start checking with the easiest axioms. So, we check the Zero axioms for all multiplication terms. If we find any unsatisfied Zero axiom, then we add it to the linearized formula and re-check. We can also check any axioms regardless of whether the axiom is the easiest one for all multiplication terms. Then, collect all unsatisfied axioms to A which are added to the linearized formula and re-check. Moreover, we can remove some unsatisfied axioms from A before adding it to the linearized formula. So, there are no specific rules to generate A. If A is not empty, we refine and continue. If A is empty, we think about the next axiom type.
We have tried different heuristics and observed the running times for evaluation. We have tried to conclude whether it is better to add always one unsatisfied axiom to A or it is better to add one for each multiplication term or some multiplication terms. We play around because we know that we will not get the same solution for different

heuristics. It is entirely up to us how we want to play with the axioms. That is why we try different ways to compute the running time and evaluate to find the best heuristic.
We have defined five types of axioms in our work: Zero axiom, Monotonicity axiom, Tangent plane axiom, Congruence axiom and ICP axiom. Each heuristic will be applied to different sequences of these axiom types for collecting unsatisfied axioms. It is up to us how we want to make different sequences of axiom types. We try to place the lightweight axiom type first in the sequences. We have used 7 different sequences of axiom types:

1. **Sequence 1:** Zero, Tangent plane, ICP, Congruence, Monotonicity.
2. **Sequence 2:** Tangent plane, Zero, ICP, Congruence, Monotonicity.
3. **Sequence 3:** Tangent plane, ICP, Zero, Congruence, Monotonicity.
4. **Sequence 4:** ICP, Zero, Tangent plane, Congruence, Monotonicity.
5. **Sequence 5:** ICP, Tangent plane, Zero, Congruence, Monotonicity.
6. **Sequence 6:** Congruence, Zero, Tangent plane, ICP, Monotonicity.
7. **Sequence 7:** Monotonicity, Zero, Tangent plane, ICP, Congruence.

Each solver instance uses one of the above sequences and maintains a pointer that initially points to the first entry in the sequence. For each refinement, the axiom type to which the pointer points to is considered for refinement and the pointer is moved to the next entry (or to the first one after considering the last entry). This is repeated until an axiom type if found that could be used for refinement, i.e., until an axiom of that type is detected to be violated by the linear solution. For the next refinement we continue at the same position in the list, i.e., we do not start each time at the beginning of the sequence.
For each of the above axiom type sequences, we use 3 different heuristics to decide how many axiom instances we want to consider for the refinement:

1. **Heuristic 1 (FIRST):** Each loop will be run over the axioms following a given fixed order (one of the 7 orders above). It will search in this order for an axiom instance that is violated by the solution for the abstraction. The first such axiom instance is added to A and the refinement process terminates.
2. **Heuristic 2 (ALL):** Similarly to 1, but when a violating axiom instance is found, then all violating instances of the given axiom are added to A.
3. **Heuristic 3 (RANDOM):** Similarly to 2 but here a percentage of unsatisfied axioms are deleted from A randomly. We have also chosen this percentage randomly from 25% to 50% of the size of A.

Notice that if we apply each heuristic over each sequence, we will get 21 different solutions for $\hat{\varphi}$. We compute running time and evaluate it in different ways. The evaluation results are discussed in Chapter 5.

Chapter 4

Implementation

4.1 SMT-RAT

We implemented our algorithm in the SMT toolbox called SMT-RAT which is entirely written in C++. Modules are the elementary architectural components of SMT-RAT. SMT-RAT is extensible: It is possible to integrate SMT-RAT with new modules which can act as a theory solver. The execution order of modules is defined in a strategy which is a directed graph. Depending on the requirement, different strategies may contain different module references in a different order. Modules can have settings arguments. So, several strategies can have the same order of modules but with different settings parameter. Settings are some configurations which can be used to configure a module in different ways without changing the code. SMT-RAT takes as input, additionally to a formula, the name of a strategy that defines which modules need to be executed in which order to decide the satisfiability of the input formula. SMT-RAT outputs SAT if a formula is satisfiable, UNSAT if a formula is unsatisfiable and UNKNOWN if satisfiability cannot be decided. Figure 4.1 shows an overview of the architecture of SMT-RAT.
We can see the toolbox has various kinds of modules for example, *SATModule*, *LRAModule*, *NRAILModule* and so on. Here, *SATModule* is a SAT solver and there are other modules which are theory solvers e.g. *LRAModule*, *NRAModule*. Besides SAT solver and theory solvers, there are other modules which are only responsible for some preprossessing, e.g. the *FPPModule*. The module *NRAILModule* is our implemented module. There are different strategies available and one of them is *NRARefinementSolver* which is our defined strategy. In strategies, the modules are connected hierarchically as a directed graph. A module in a strategy can only use its direct successor modules for computation because a module may need the help of another module. A module is addressed as backend and frontend for its direct ancestor and successor modules, respectively. Note that a module does not know which module is its successor or ancestor module.

4.2 NRARefinementSolver

The class *NRARefinementSolver* is a strategy where the order of the modules is declared hierarchically as shown in Figure 4.1. The first module of this strategy is

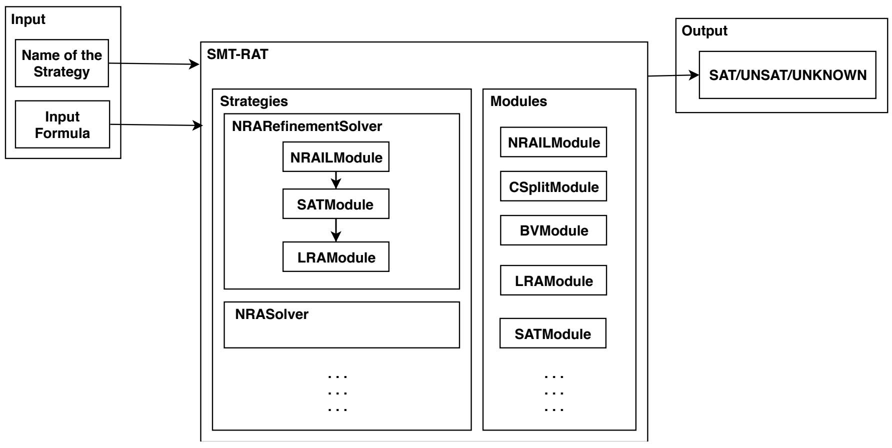


Figure 4.1: Overview of the architecture of SMT-RAT

NRAILModule which implements incremental linearization for real arithmetic as described in Chapter 3. The next module *SATModule* creates a boolean skeleton of formula and solves the resulting formula with the SAT solver *MiniSat*, where after each completed decision level the constraints belonging to the assigned boolean variables are checked for consistency by the backend *LRAModule* of the *SATmodule* [CKJ$^+$15]. If *LRAModule* finds inconsistency, then it provides an unsatisfiable core which is abstracted and participated in search for a satisfying solution. The *LRAModule* implements an SMT compliant *Simplex* method which is a method for solving LRA constraint sets [CKJ$^+$15]. In the following we will explain the significant implementation details of the module *NRAILModule*.

4.2.1 NRAILModule

In order to be a module of SMT-RAT, the class *NRAILModule* has to be a child of class *Module*. A basic module can implement the functions *informCore*, *addCore*, *removeCore*, *updateModel* and *checkCore*. Our module implements the functions *addCore* and *checkCore* whose algorithms are briefly described in the Sections 3.2.1 and 3.2.2, respectively.

4.2.1.1 Mapping of z-variable to Multiplication Term

In Section 3.2.1.2 we have seen how a monomial m is abstracted by z-variables (i.e., $z_1, z_2, \ldots, z_i$ for some $i \in \mathbb{N}$) following Algorithm 6. During the abstraction, it is mandatory to keep track of the z-variables and their corresponding encapsulated multiplication terms (i.e., $x * x$, $x * y$ and so on). There are two reasons to keep track. Firstly, each z-variable and its corresponding encapsulated multiplication term are used to construct formulas for different axiom types. Secondly, we want to prevent the recreation of different z-variables for the same multiplication term. At the beginning, we kept each z-variable and its corresponding encapsulated multiplication term in a pair data structure and collected these pairs in a list data structure for some $i \in \mathbb{N}$:

$$pair_i = [z_i, term_i]$$

$$List = [pair_1, pair_2, \ldots, pair_k]$$

To create formulas for axioms, it is enough to iterate over the pairs in the list. In our thesis work, we estimated assignments for an input NRA formula φ (see line 8, Algorithm 9). However, in future, we want to create NRA model and for that, we need to know which term is encapsulated in each z-variable to reach the original terms of φ (see Equation 3.1, Example 3.2.2). That means we have to search over the list. So, we have to implement a search operation for searching multiplication terms abstracted by z-variables, whereas map comes with the search by key out of the box. Thus, we use a map to keep the track. Map stores key-value pairs and we keep each z-variable as a key and its corresponding abstracted multiplication term as value:

$$Map = [(z_1, term_1), (z_2, term_2), \ldots, (z_k, term_k)]$$

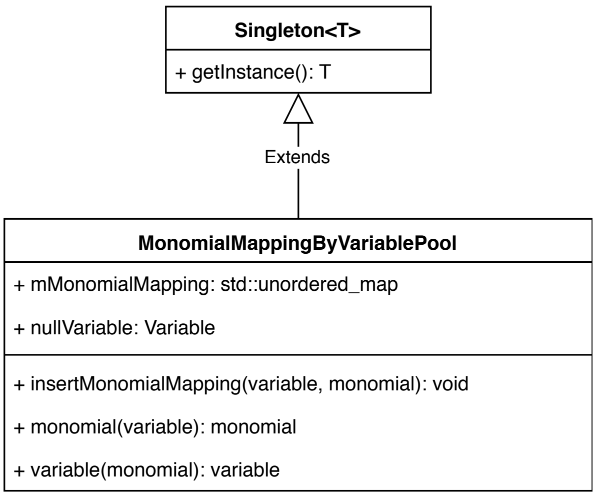

Singleton<T>
+ getInstance(): T
Extends
MonomialMappingByVariablePool
+ mMonomialMapping: std::unordered_map
+ nullVariable: Variable
+ insertMonomialMapping(variable, monomial): void
+ monomial(variable): monomial
+ variable(monomial): variable

Figure 4.2: Class diagram of *mMonomialMapping*

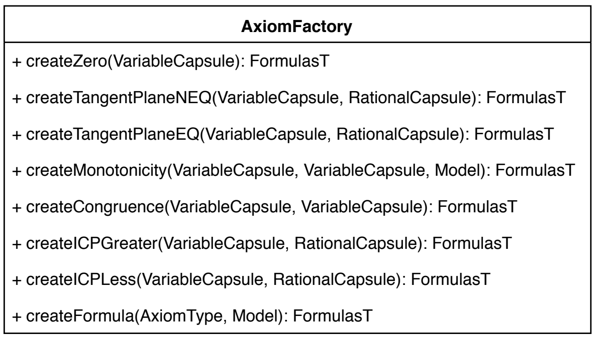
From Figure 4.2, we can see there is an internal map *mMonomialMapping* where we keep the z-variables and multiplication terms. We also declared a special variable called *nullVariable* which we used to express that a monomial has no abstraction variable in the map (see below). The function *insertMonomialMapping* takes a z-variable and corresponding multiplication term as monomial and then insert them into the internal map. The function *monomial* takes a z-variable and searches it in the *mMonomialMapping.* If it finds the z-variable, then it returns the corresponding multiplication term as a monomial. Otherwise, it returns a null pointer. Similarly, the function *variable* takes a multiplication term as monomial and searches it in the *mMonomialMapping.* If it finds the monomial, then it returns the z-variable. Otherwise, it returns the *nullVariable.*

4.2.1.2 Generation of Axiom Formulas

AxiomFactory is a class that creates formulas for different axiom types. We already know that there are five types of axioms in our work: Zero axiom, Monotonicity axiom, Tangent plane axiom, Congruence axiom and ICP axiom. We have defined a data type for each type of axiom in the enum class *AxiomType* as follows:

Axiom Type	Data Type
Zero	*ZERO*
Monotonicity	*MONOTONICITY*
Tangent plane	*TANGENT_PLANE*
Congruence	*CONGRUENCE*
ICP	*ICP*

Figure 4.3 shows the class diagram of *AxiomFactory*. The class *AxiomFactory* has a function *createFormula* that creates axiom formulas depending on the parameter *AxiomType*. The function has another parameter of *Model* type that is a model *linearizedModel* containing the satisfying assignments for the elements of the map *mMonomialMapping*. We have mentioned earlier in Section 3.2.3 that the linearized model can be partial for some multiplication terms which are stored in the map *mMonomialMapping*; in this case we guess the missing values for the variables in those multiplication terms. Therefore, we combine the linearized model *mModel* with the guessed values for the rest of the variables into the *linearizedModel* and then it is passed to the function *createFormula*.
We create two Data Transfer Objects called *VariableCapsule* and *RationalCapsule*. *VariableCapsule* and *RationalCapsule* encapsulate three *Variable* objects and *Rational* objects, respectively. We introduce these data transfer objects to avoid long parameter lists.
We have defined different functions to generate formulas for different axiom types in *AxiomFactory*. We use SMT-RAT APIs to create the formulas in these functions. If *AxiomType* is *ZERO* or *TANGENT_PLANE* or *ICP*, we extract variables from each map element and the extracted variables will be encapsulated in *VariableCapsule*. Afterwards, we use this *VariableCapsule* to generate formulas for the passed *AxiomType*. Remember that the list of different types of axioms is shown in Figure 3.1. When the *AxiomType* is *ZERO*, we only pass *VariableCapsule* to create axiom formulas because we do not need assignments of the variables encapsulated in the *VariableCapsule*. Otherwise, for *AxiomType* *TANGENT_PLANE* or *ICP*, we pass both *VariableCapsule* and *RationalCapsule* to create axiom formulas.


AxiomFactory

+ createZero(VariableCapsule): FormulasT
+ createTangentPlaneNEQ(VariableCapsule, RationalCapsule): FormulasT
+ createTangentPlaneEQ(VariableCapsule, RationalCapsule): FormulasT
+ createMonotonicity(VariableCapsule, VariableCapsule, Model): FormulasT
+ createCongruence(VariableCapsule, VariableCapsule): FormulasT
+ createICPGreater(VariableCapsule, RationalCapsule): FormulasT
+ createICPLess(VariableCapsule, RationalCapsule): FormulasT
+ createFormula(AxiomType, Model): FormulasT

Figure 4.3: Class diagram of *AxiomFactory*

If the *AxiomType* is *MONOTONICITY* or *CONGRUENCE*, then similarly, we iterate over the map, but for each iteration, we iterate over the map again by a nested loop. So, we have one outer and one inner loop. Then, we have extracted variables to be encapsulated in *VariableCapsule* objects for the outer and inner loop. Here, the first and the second parameter of the functions *createMonotonicity* and *createCongruence* represent the corresponding *VariableCapsule* objects as outer and inner loop, respectively. The function *createMonotonicity* also has a third parameter of *Model* type. The implementation of *createMonotonicity(VariableCapsule, VariableCapsule, Model)* is different from other formula creator functions and we will explain the implementation details in the next section.

4.2.1.3 Generation of Formulas for Monotonicity

From Figure 3.1, we can see that we have to generate axiom formulas for Monotonicity axiom with the absolute value of the variables i.e., $abs(x_1)$, $abs(y_1)$, $abs(z_1)$ and so on, for each multiplication term. There is no built-in function in SMT-RAT to create a formula with absolute value. To solve this problem, first we create an equivalent formula for the absolute value of each variable x by introducing a new auxiliary variable aux_x for x as follows:

$$abs(x) := (x \geq 0 \rightarrow aux\_x = x) \quad \wedge \quad (x < 0 \rightarrow aux\_x = -x)$$

Then as shown in Figure 4.4, we create equivalent formulas for each constraint i.e., $abs(x_1) \leq abs(x_2)$, $abs(y_1) \leq abs(y_2)$, $abs(z_1) \leq abs(z_2)$ and so on, of the original Monotonicity axiom formulas (Figure 3.1). Note that in Figure 4.4, the relation between two auxiliary variables depends on the relation between the absolute values of the two variables of the constraint. Finally, connect the equivalent formulas according to their connecting logical operators.

$$\begin{aligned} abs(x_1) \leq abs(x_2) := \quad & x_1 \geq 0 \rightarrow aux\_x_1 = x_1 && \wedge \\ & x_1 < 0 \rightarrow aux\_x_1 = -x_1 && \wedge \\ & x_2 \geq 0 \rightarrow aux\_x_2 = x_2 && \wedge \\ & x_2 < 0 \rightarrow aux\_x_2 = -x_2 && \wedge \\ & aux\_x_1 \leq aux\_x_2 \end{aligned}$$

Figure 4.4: Equivalent formula for a constraint with absolute value of variables

In Section 3.2.3, we have seen how we find the unsatisfied axiom formulas to be added to the linearized formula. The axiom formulas are called unsatisfied axiom formulas if they are not satisfied by the linearized model. But for *AxiomType MONOTONICITY* we cannot find the unsatisfied formulas until the linearized model has the assignments for the auxiliary variables. That is why we thought to create such a model which contains the assignments for auxiliary variables. For this, we had to guess a value for each auxiliary variable. We wanted to try two different values for each auxiliary variable (i.e., $aux\_x_1 = x_1$ or $aux\_x_1 = -x_1$ and $aux\_x_2 = x_2$ or $aux\_x_2 = -x_2$) containing in a constraint to check if the considered values satisfy the equivalent formula of this constraint. If we find such a value, then we will assign this as a value to the auxiliary variable. Otherwise, we can choose any of two different values. So, for each constraint, we have to try out four different combinations of values for the two auxiliary variables. This solution can make our refinement process more costly and the performance may decrease a lot. That is why we do not pick this solution instead think about other optimized solution.

We create a model *absoluteValuedModel* which contains the absolute value of variables of the model *linearizedModel*. This *absoluteValuedModel* is passed as the third parameter of the function *createMonotonicity* which was mentioned in the previous section. The function *createMonotonicity* creates Monotonicity axiom formulas as shown in Figure 3.1 for a pair of multiplication terms and checks if these are unsatisfied by *absoluteValuedModel*. If unsatisfied formulas are found, then their equivalent formulas are generated and accumulated into the *List monotonicityFormulas*. Once all unsatisfied formulas are collected for that pair of multiplication terms, the function *createMonotonicity* returns *monotonicityFormulas* to its caller method.

4.2.1.4 Module Settings

We apply each possible combination of three heuristics and seven sequences of axiom types in the refinement process to find less costly pair of heuristic and sequence. We need to declare them somewhere so that we can choose any combination easily. SMT-RAT provides *Settings* API by which we can declare various settings and afterward we have to set the required settings in the strategy as a module parameter. At run-time, we can read the settings from the module and perform the selected settings specific operation. So, we have to declare 21 different settings namely *NRAILSettings*1, *NRAILSettings*2, ..., *NRAILSettings*21.
Now we will explain how we declare our different settings for *NRARefinementSolver*. Each setting must point to a heuristic and a sequence of axiom types. We declare an enum class *UNSATFormulaSelectionStrategy* to represent the heuristics. This class has three enum types for three heuristics: *FIRST* for heuristic 1, *ALL* for heuristic 2 and *RANDOM* for heuristic 3. Each setting has one of these defined enum types. Then we declare a sequence of axiom types in an array by putting the enum types of enum class *AxiomType* sequentially. This array contains one of the sequences (Sequence 1, ..., Sequence 7). So, the refinement process is performed over the axiom types in the same sequence as decelerated in the selected settings and collects unsatisfied formulas according to defined heuristic in those settings. We used *NRAILSettings*1 *as specified in Figure 4.1 as our default settings.*

Listing 4.1: Default settings

```
struct NRAILSettings1
{
 moduleName = "NRAILModule<NRAILSettings1>";

 UNSATFormulaSelectionStrategy
 formulaSelectionStrategy = UNSATFormulaSelectionStrategy::ALL;

 AxiomFactory::AxiomType
 axiomType[5] = {AxiomFactory::AxiomType::ZERO,
                 AxiomFactory::AxiomType::TANGENT_PLANE,
                 AxiomFactory::AxiomType::ICP,
                 AxiomFactory::AxiomType::CONGRUENCE,
                 AxiomFactory::AxiomType::MONOTONICITY};
};
```

As we have 21 different settings, we have also created 21 different strategies namely *NRARefinementSolver*1, *NRARefinementSolver*2, ..., *NRARefinementSolver*21 with *NRAILSettings*1, *NRAILSettings*2, ..., *NRAILSettings*21, respectively. Each of them are copies of *NRARefinementSolver* shown in Figure 4.1 but with corresponding settings as module parameter. The strategy *NRARefinementSolver* has the default settings.

Chapter 5

Experimental Results

5.1 Experimental Setup

We evaluate our implementation by running some benchmarks over different strategies as well as solvers. We have used QF_NRA benchmarks [SMT16] from SMT-LIB [BFT16]. We run the benchmarks on a cluster having multiple processors of 2.10 GHz Intel Xeon Platinum 8160 and 8GB of memory per process. We decided to use a timeout of 120 seconds (s) per problem instance for the experiment.

5.2 Results

We have already described in Section 4.2.1.4 that we use different strategies *NRARefinementSolver*1, *NRARefinementSolver*2, ..., *NRARefinementSolver*21 which are mentioned as smtrat1, smtrat2, ..., smtrat21, respectively in this chapter. The results for smtrat1, smtrat2, ..., smtrat21 without preprocessing (WoP) are reported in Table 5.1 where the first column contains the solver's name. The second and third column contains the number of satisfiable and unsatisfiable instances that could solve within the time limit, respectively, with average solving time. The total number of solved instances with the percentage of the solution is reported in the last column from worst to best. The detailed settings of each solver are also provided in Table 5.2 by a pair of heuristic type and sequence of axiom types (Section 4.2.1.4).
Table 5.1 shows some interesting aspects. The solver smtrat4 performs the best, whereas smtrat7 performs the worst. However, smtrat11 and smtrat3 solve the highest number of satisfiable and unsatisfiable instances, respectively. It is noticeable that six solvers of heuristic type *ALL* which is marked green perform the best in a row. On the other hand, the rows marked red highlight all solvers of heuristic type *RANDOM* perform worse than other solvers in a row. Also, this group of solvers solves less unsatisfiable instances compared to other solvers. We can see that smtrat7, smtrat14 and smtrat21 share the same sequence of axiom types from Table 5.2. Interestingly, two solvers smtrat7 and smtrat21 (both are marked black) among these three solvers rank at the last which demonstrates that sequence 7 is not effective because it has the most expensive axiom Monotonicity at the very beginning of the sequence. The sequence 4 is followed by both our best solver smtrat4 and the highest number of satisfiable instances solving solver smtrat11. Moreover, smtrat4 and smtrat11 take less

Table 5.1: Number of solved instances for our solvers without preprocessing (WoP)

Solver	**SAT**		**UNSAT**		**Overall**	
smtrat7	1678	2.53 s	3806	1.88 s	5484	47.7 %
smtrat21	1735	3.06 s	3759	1.91 s	5494	47.8 %
smtrat13	1867	1.12 s	3659	1.12 s	5526	48.1 %
smtrat14	1893	1.68 s	3646	1.19 s	5539	48.2 %
smtrat9	1899	1.39 s	3659	1.30 s	5558	48.4 %
smtrat10	1895	1.54 s	3664	1.30 s	5559	48.4 %
smtrat12	1902	1.61 s	3657	1.14 s	5559	48.4 %
smtrat11	1916	1.21 s	3663	1.17 s	5579	48.6 %
smtrat8	1914	1.39 s	3669	1.24 s	5583	48.6 %
smtrat16	1801	1.94 s	3818	1.84 s	5619	48.9 %
smtrat20	1813	1.70 s	3810	1.83 s	5623	48.9 %
smtrat17	1809	1.99 s	3827	1.92 s	5636	49.1 %
smtrat19	1831	1.83 s	3816	1.83 s	5647	49.2 %
smtrat18	1817	2.28 s	3840	2.10 s	5657	49.2 %
smtrat15	1859	2.34 s	3826	2.04 s	5685	49.5 %
smtrat6	1811	1.84 s	3911	1.99 s	5722	49.8 %
smtrat5	1828	1.94 s	3897	1.76 s	5725	49.8 %
smtrat1	1820	1.79 s	3908	1.61 s	5728	49.9 %
smtrat2	1827	1.99 s	3902	1.75 s	5729	49.9 %
smtrat3	1810	1.73 s	3924	2.17 s	5734	49.9 %
smtrat4	1849	1.64 s	3914	1.67 s	5763	50.2 %

solving time on average for both instances than most of the solvers. So, it can be said that sequence 4 is the most effective sequence where our defined axiom ICP is placed at the beginning and ICP helps to reach the solution more quickly. It also can be said that a pair of heuristic type ALL and sequence 4 helps to improve the performance of a solver. Therefore, we decide to consider smtrat4 for further evaluation which follows heuristic type ALL to collect unsatisfiable axioms by performing refinement over sequence 4.

Table 5.3 has the same structure as Table 5.1. Here, we compare smtrat4 with other two SMT solvers Z3 [dMB08] and MathSAT [CGSS13]. We have taken two versions of smtrat4 which are without preprocessing (WoP) and with preprocessing (WP). In order to compare the heuristics without external influence of other modules, we switched off preprocessing. The solvers smtrat4 WoP and WP both get the same original inputs but the former is without and the latter is with preprocessing help. So, for compatibility, we add also a version WP. It is visible that Z3 and MathSAT have much better performance than smtrat4 in all cases. The reason is that the SMT solver Z3 implements expensive and complete techniques based on variants of the cylindrical algebraic decomposition method [CGI^{+}18]. We are rather interested in comparing smtrat4 with MathSAT as they implement similar approaches. The SMT solver MathSAT implements the incremental approach as described in [Irf18] which was a Ph.D. thesis; we have implemented a slightly modified version of that approach in our thesis, but our implementation is rather prototypical. Furthermore, our algorithm does not contain some effective components that are implemented in

Table 5.2: Settings of our solvers

Solver	Heuristic type	Sequence no / Sequence of axiom types
smtrat1	ALL	1 / Zero, Tangent plane, ICP, Congruence, Monotonicity
smtrat2		2 / Tangent plane, Zero, ICP, Congruence, Monotonicity
smtrat3		3 / Tangent plane, ICP, Zero, Congruence, Monotonicity
smtrat4		4 / ICP, Zero, Tangent plane, Congruence, Monotonicity
smtrat5		5 / ICP, Tangent plane, Zero, Congruence, Monotonicity
smtrat6		6 / Congruence, Zero, Tangent plane, ICP, Monotonicity
smtrat7		7 / Monotonicity, Zero, Tangent plane, ICP, Congruence
smtrat8	RANDOM	1 / Zero, Tangent plane, ICP, Congruence, Monotonicity
smtrat9		2 / Tangent plane, Zero, ICP, Congruence, Monotonicity
smtrat10		3 / Tangent plane, ICP, Zero, Congruence, Monotonicity
smtrat11		4 / ICP, Zero, Tangent plane, Congruence, Monotonicity
smtrat12		5 / ICP, Tangent plane, Zero, Congruence, Monotonicity
smtrat13		6 / Congruence, Zero, Tangent plane, ICP, Monotonicity
smtrat14		7 / Monotonicity, Zero, Tangent plane, ICP, Congruence
smtrat15	FIRST	1 / Zero, Tangent plane, ICP, Congruence, Monotonicity
smtrat16		2 / Tangent plane, Zero, ICP, Congruence, Monotonicity
smtrat17		3 / Tangent plane, ICP, Zero, Congruence, Monotonicity
smtrat18		4 / ICP, Zero, Tangent plane, Congruence, Monotonicity
smtrat19		5 / ICP, Tangent plane, Zero, Congruence, Monotonicity
smtrat20		6 / Congruence, Zero, Tangent plane, ICP, Monotonicity
smtrat21		7 / Monotonicity, Zero, Tangent plane, ICP, Congruence

MathSAT, like the adaptation of solutions for the linearization to satisfy the non-linear problem, or their piecewise linear refinement technique for concave solution spaces.

The solver smtrat4 WP solves above 2.3K satisfiable and 4.3K unsatisfiable instances but smtrat4 WoP solves below 2K satisfiable and 4K unsatisfiable instances. So, the performance of smtrat4 is increased by 8% while switching on preprocessing. Also, smtrat4 performs much better for unsatisfiable benchmarks than satisfiable benchmarks for both cases whether switching off or on preprocessing. We can see that MathSAT solves above 3.5K satisfiable and 5.2K unsatisfiable instances. This is 27% more than smtrat4 WoP and 19% more than smtrat4 WP. Remember that our solver outputs only UNSAT if the LRA formula is found unsatisfiable by the SMT solver and SAT if the input NRA formula φ is satisfied by the model μ of its linear abstraction (Figure 3.1). Here, we use a different theory solver than MathSAT to solve LRA formulas. Furthermore, we only extend the LRA model $\hat{\mu}$ with values for variables that do not occur in the linearization, but we did not implement the repair. On the contrary, MathSAT tried to repair $\hat{\mu}$ which might help. Furthermore, we refine the abstraction using different axioms and different heuristics.

Figure 5.1 shows survival plots for two versions of smtrat4, MathSAT and Z3. The horizontal axis shows the number of instances solved within the corresponding time and the vertical shows the runtime in seconds. The survival plots behave exponentially. The solver smtrat4 WoP solves the least number of instances and Z3 solves the highest number of instances within the time limit. However, smtrat WP and MathSAT solve approximately 6.7K and 8.9K instances in total. Initially, both of

Table 5.3: Comparison of smtrat4 with MathSAT and Z3

Solver	**SAT**		**UNSAT**		**overall**	
smtrat4 WoP	1849	1.64 s	3914	1.67 s	5763	50.2 %
smtrat4 WP	2336	1.55 s	4348	2.41 s	6684	58.2 %
MathSAT	3560	1.20 s	5286	1.80 s	8846	76.9 %
Z3	5044	2.14 s	5099	1.63 s	10076	87.9 %

the smtrat4 solvers start solving instances earlier than Z3 followed by MathSAT. The performance of smtrat4 decreases later.
There are two scatter plots shown in the Figures 5.2 and 5.3. We have generated these scatter plots to compare pairwise solvers on individual benchmark instances. Each point (x, y) in a scatter represents a benchmark instance which was solved by a solver in time x and another solver in time y. Points on the inner edges labeled by T indicate timeouts, whereas on the outer edges labeled by M indicate memory outs. We have chosen smtrat by switching on preprocessing to draw the scatter plots because this solver performs better than switching off preprocessing.

Figure 5.1: Survival plots for smtrat4 WoP, smtrat4 WP, MathSAT and Z3

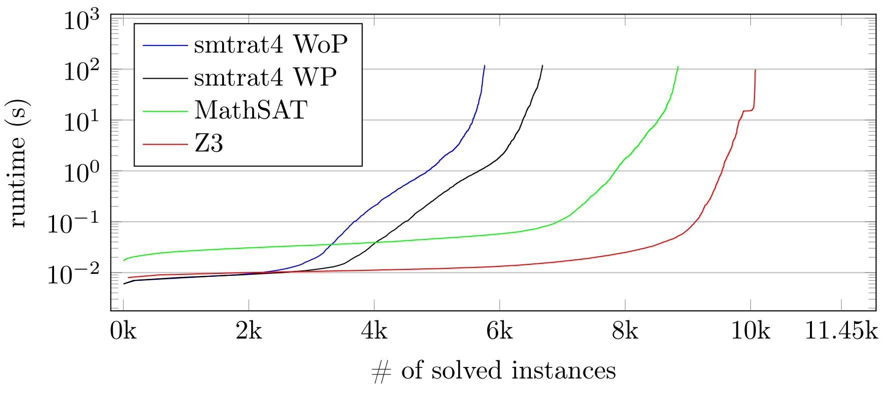


Figure 5.2 shows the scatter plot for smtrat4 WP with MathSAT. We can see a thick cloud at (0, 0) which means that smtrat4 WP and MathSAT solve many instances quickly. Then a thinner cloud extends linearly titled to the bottom especially near the horizontal axis. So, MathSAT is faster than smtrat4 WP for most of the instances. There is also a branch near the vertical axis; thus smtrat4 WP is also faster than MathSAT for some instances. Besides, the dots on the edge T parallel to the vertical axis implies the instances which were solved by MathSAT, but smtrat4 WP was unable to solve those. Similarly, the dots on the edge T parallel to the horizontal axis implies the opposite phenomena though the dots are in less number compared to the edge T parallel to the vertical axis. We can see different behavior in Figure 5.3 which is the scatter plot for smtrat4 WP and Z3. Here, the cloud is distributed. There is a cloud splits with one branch near the horizontal axis and another branch near the vertical axis. There are also some dots that are a bit tilted to the top or the bottom of the linear line. It means that both solvers behave similarly to these instances. The solver Z3 could also solve a massive number of instances for which smtrat4 WP results in

timeouts.

Figure 5.2: Scatter plot for smtrat4 WP and MathSAT

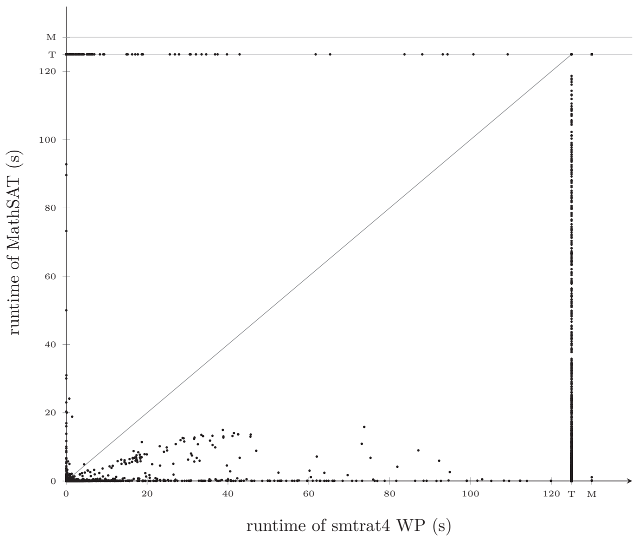
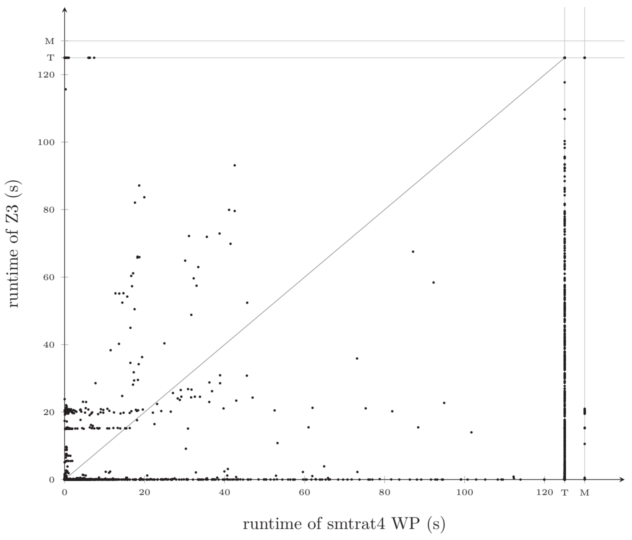


Figure 5.3: Scatter plot for smtrat4 WP and Z3



We already know that we have defined three heuristic types and seven sequences of axiom types. We have also seen that the pair of heuristic type ALL and sequence 4 can enhance the performance of a solver. In other words, ICP has an influence on the enhancement of the performance. That is why we think about to play around with the sequence by concentrating on ICP the highest. So, for experimental purpose we create four additional SMT-RAT solvers (smtrat22, . . . , smtrat25) by switching on preprocessing with the following sequences of axiom types but with the same heuristic type ALL:

- smtrat22 -> ICP, Zero, Tangent plane, Congruence
- smtrat23 -> ICP, Tangent plane, Zero, Congruence
- smtrat24 -> ICP, Zero, ICP, Tangent plane, ICP, Congruence
- smtrat25 -> ICP, Tangent plane, ICP, Zero, ICP, Congruence

Table 5.4: Summary of smrat4, other additional smtrat solvers, MathSAT and Z3

Solver	**SAT**		**UNSAT**		**overall**	
smtrat4 WP	2336	1.55 s	4348	2.41 s	6684	58.2 %
smtrat23 WP	2438	0.61 s	4449	2.31 s	6887	59.9 %
smtrat25 WP	2401	0.87 s	4490	2.50 s	6891	60.0 %
smtrat22 WP	2465	0.63 s	4443	2.39 s	6908	60.1 %
smtrat24 WP	2463	0.92 s	4463	2.40 s	6926	60.3 %
MathSAT	3560	1.20 s	5286	1.80 s	8846	76.9 %
Z3	5044	2.14 s	5099	1.63 s	10076	87.9 %

We exclude the axiom type Monotonicity from the sequence for each of these solvers as Monotonicity is the most costly to generate. The results are reported in Table 5.4 which maintains the same structure as Tables 5.1 and 5.3. Compared to smtrat4 WP, the total number of solved instances has increased the most (2%) for smtrat24 WP. As a result, the difference between the overall performances of our solver and MathSAT has decreased from 19% to 17%. Notice that smtrat24 follows almost the same pattern of the sequence 4 except that ICP is inserted after each different axiom type in its sequence. Figures 5.4 and 5.5 are the scatter plots for smtrat24 WP with MathSAT and Z3, respectively. These two scatter plots have the same pattern as the scatter plots for smtrat4 WP with MathSAT (Figure 5.2) and Z3 (Figure 5.3), respectively.

Figure 5.4: Scatter plot for smtrat24 WP and MathSAT

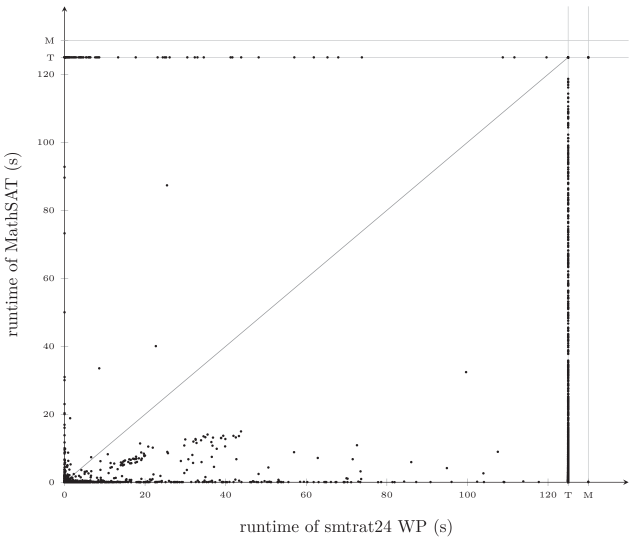
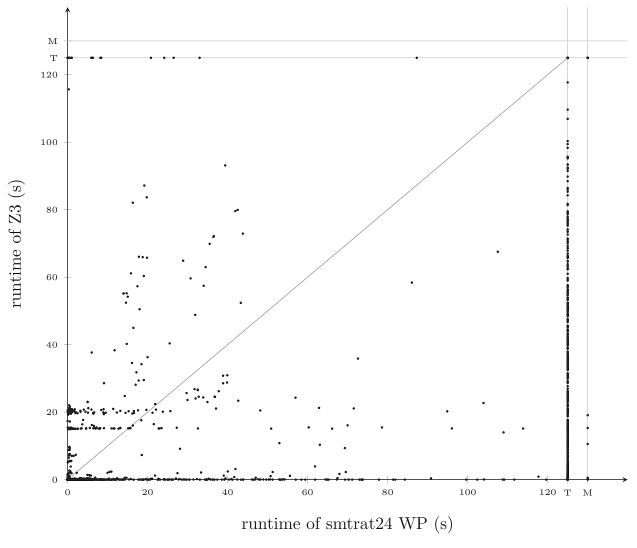


Figure 5.5: Scatter plot for smtrat24 WP and Z3



Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis work focuses on how we can decide the satisfiability of logical formulas. Solving NRA formulas is costly compared to LRA formulas. For the ease of solving NRA formulas, we perform abstraction. This thesis work is motivated by the Ph.D. thesis [Irf18, CGI^{+}18] and the idea is to create a LRA formula for the input NRA formula by abstracting multiplication terms iteratively. If the LRA model does not satisfy the NRA formula, then refinement is performed over a set of axioms to remove spurious assignments until we get SAT or UNSAT or UNKNOWN. Moreover, we have introduced three types of heuristics with different sequences of axioms for collecting unsatisfiable formulas which are added to the LRA formula as refinement. We have also introduced ICP axioms for refinement which is one of the future work of [CGI^{+}18]. We have integrated this method as a module in SMT-RAT. We have evaluated the performance of the module in Chapter 5. We have seen that the module can solve 53% instances in total. For now, the percentage is satisfactory as this module is a prototype and there are scopes available to improve the module which is mentioned in the following section. We were unable to implement these due to time limitations. We expect a significant enhancement of the performance once the mentioned future works will be integrating with the module.

6.2 Future Work

Due to time limitation, we could not complete some essential tasks which are needed to make the solver more stable. In the future, we need more time to invest in the following areas:

- We have already several axioms, but there are still further possibilities to add axioms. Currently, we can exclude only boxes by ICP. However, it is also possible to exclude regions of different shapes based on templates. That means we can try other forms of areas which we want to exclude and then derive axioms for them.

- So far, we have assigned the values of original variables by guessing if the linearized model $\hat{\mu}$ does not have solutions for all original variables which may results in UNSAT for satisfiable benchmarks. So, it is essential to generate solutions to original variables as well as other variables. For example, consider the following formula:

$$\varphi := x^4y^2 - y > 5$$

The monomial x^4y^2 is linearized as follows and we get the linearized formula $\hat{\varphi} := z_4 - y > 5$:

$$\underbrace{\underbrace{\underbrace{x * x}_{z_1} * \underbrace{x * x}_{z_1}}_{z_2} * \underbrace{y * y}_{z_3}}_{z_4}$$

Now, $\hat{\varphi}$ is passed to the SMT LRA solver and the solver outputs SAT with linearized model $\hat{\mu} := \{z_4 = 16, y = 1.414\}$. Notice that $\hat{\varphi}$ does not cover all original variables. Currently, we extend $\hat{\mu}$ by guessing zero for x as well as other variables z_1, z_2 and z_3 which will in most cases lead us to the refinement process. Hence, instead of assuming values we can find out the solutions for these variables based on $\hat{\mu}$ (if available):

- ⋆ $z_4 = z_2 * z_3 = z_2 * y * y$. So, $z_2 = 4$.
- ⋆ $z_1 = 2$ as $z_2 = z_1^2$.
- ⋆ Finally, $x = 4$.

In the future, we will integrate this feature to our solver which will resist going through the refinement process unnecessarily for satisfiable benchmarks. Most importantly this feature will enhance the solver's performance by solving more satisfiable instances.

Bibliography

[Ábr] Erika Ábrahám. Full lazy SMT-solving. https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/vorlesung_satchecking/ws14_15/05a_smt_handout.pdf. Last accessed 31 Oct 2018.

[Bar] Clark Barrett. Theory solvers. https://web.stanford.edu/class/cs357/lectures/lec10.pdf. Last accessed 31 Oct 2018.

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

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[CGI$^+$18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. *ACM Trans. Comput. Logic*, 19(3):1–52, 2018.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2013*, pages 93–107. Springer, 2013.

[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 *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 360–368. Springer, 2015.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification – CAV 2006*, pages 81–94. Springer, 2006.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2008*, pages 337–340. Springer, 2008.

[dMDS07] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Computer Aided Verification – CAV 2007*, pages 20–36. Springer, 2007.

[Har] Gregory Hartman. Tangent Lines, Normal Lines, and Tangent Planes. `http://spot.pcc.edu/math/APEXCalculus/sec_multi_tangent.html`. Last accessed 08 Oct 2018.

[Irf18] Ahmed Irfan. *Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions.* PhD thesis, University of Trento and Fondazione Bruno Kessler, 2018.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[SMT16] The Satisfiability Modulo Theories Library (SMT-LIB) - Benchmarks. `http://smtlib.cs.uiowa.edu/benchmarks.shtml`, 2016.