SMT for Polynomial Constraints and Its Applications

by

TO VAN KHANH

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Mizuhito Ogawa

School of Information Science
Japan Advanced Institute of Science and Technology

June, 2013

Abstract

Solving polynomial constraints plays an important role in program verification, e.g., checking roundoff and overflow errors with fixed point or floating point arithmetic, measures for proving termination, and linear loop invariant generation. Tarski proved that polynomial constraints over real numbers (algebraic numbers) are decidable, and later Collins proposed Quantifier Elimination by Cylindrical Algebraic Decomposition, which is nowadays implemented in Mathematica, Maple/SyNRAC, Reduce/Redlog, and QEPCAD. However, it is DEXPTIME with regard to the number of variables, and works fine in practice up to 5 variables and lower degrees. For instance, 8 variables with degree 10 may require 20–30 hours by a supercomputer.

Motivated from numerous applications of polynomial constraint solving, this thesis aims to propose an approach and develop an *SMT solver* for *solving polynomial constraints*. First, we focus on *polynomial inequality constraints* coming from following reasons.

(a) In constructive analysis, solving equality constraints on real numbers is in general undecidable (decidable only for algebraic numbers), whereas solving inequality is decidable. In other words, $a > b$ is computable, whereas $a = b$ is not computable.

(b) Inequality allows approximations.

(c) Solving polynomial inequality on real numbers is reduced to that on rational numbers. The reduction to rational numbers allows avoiding roundoff-errors in implementations.

Our approach and contributions in the thesis are summarized as follows:

(i) We propose an approach of *iterative approximation refinement* for solving constraints, which is formalized as an *abstract DPLL(T) procedure* for *over/under-approximations* and *refinements* under a background theory T. An under approximation is sound for proving in the background theory T, and an over approximation is sound for disproving. When they neither prove nor disprove, *refinements* are applied to decompose an atomic formula of the input formula, i.e., ψ to $\psi_1 \vee \psi_2$ such that $\psi \Leftrightarrow \psi_1 \vee \psi_2$. The proposed approach combined DPLL(T) procedure with over/under-approximations and refinements is sound and complete for solving polynomial inequality constraints under certain restrictions.

(ii) We instantiate *interval arithmetic* to over approximation and *testing* to under approximation. A new form of affine interval, called *Chebyshev Affine Interval*, is proposed. Chebyshev Affine Interval has an advantage over current *affine intervals* such that it can keep sources of computation for high degree variables, which would be useful for guiding refinements.

(iii) The proposed approach is implemented as the SMT solver **raSAT**, which applies *interval arithmetic* (over-approximation, aiming to decide unsatisfiability), *testing* (under-approximation, aiming to decide satisfiability), and *refinements* on interval decompositions.

(iv) We propose UNSAT cores of polynomial constraints that can improve efficiency in theory propagation of SMT. Computation of UNSAT cores in polynomial constraints allows inferring other unsatisfiable domain when a particular domain is detected as unsatisfiable. We propose an approach for incremental test data generation which would be useful when performing a large number of test data (i.e., a large number of variables).

(v) We propose strategies for refinements such that choices of intervals to decompose and methods to decompose an interval into smaller intervals. These strategies are guided from interval arithmetic, testing results, test data, and polynomials.

(vi) The proposed approach is also extended for *greater-than-or-equal* ($\geq$) constraints, i.e., $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > 0$ for proving satisfiability, and for proving unsatisfiability $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > -\delta_i$ for $\delta_i > 0$.

(vii) We propose a non-constructive method for solving polynomial constraints including *equalities* based on *intermediate value theorem*.

Key words: interval arithmetic, affine arithmetic, SAT Modulo Theories - SMT, polynomial constraints, testing, abstract DPLL.

To my wife, Nguyen Kieu Phuong, my son, To Van Nhat Minh, and my parents

Acknowledgements

I would like to express my gratitude to all those who provided me the possibility to complete this thesis.

Firstly, I would like to express my sincere gratitude to my supervisor, Professor Mizuhito Ogawa, for his supervision, advice, assistance, and guidance during the whole period of my doctoral course. Professor Mizuhito Ogawa always gave me useful comments to clearly understand problems and asked me to have a deep view on doing research, which was very useful for looking at new directions and further investigating in my field. These helped me greatly to improve my skills and make much progress. He also provided me kind encouragement and support not only in my research but also in my life.

I would like to thank the members in the Ogawa Laboratory for their help, especially Associate Professor Nao Hirokawa for his guidance and advice during my studying in Japan. He is very zealous in discussion and sharing ideas. I received a lot of valuable suggestions from him for this thesis.

I would like to acknowledge the Ministry of Education and Training - Vietnam (MOET) for their finance support of my study time in Japan. Without the support of MOET, I could not complete the doctoral course in Japan.

I would also like to acknowledge University of Engineering and Technology (UET), Vietnam National University, Hanoi, and Japan Advanced Institute of Science and Technology (JAIST), who gave me chance to join the FIVE-JAIST program and helped me to complete the program. My colleagues in UET have provided me a good environment during my studying in UET. Their support and encouragement helped me much to complete this thesis. I would like to thank my co-supervisor in UET, Associate Professor Nguyen Viet Ha, for his advice, kind support, and encouragement.

My wife, Nguyen Kieu Phuong always encouraged me to focus on doing my research and took care of me. I also want to thank my little son To Van Nhat Minh. Seeing him growing up everyday made me feel happy and helped me to release stress.

Lastly, I would like to give my special thanks to my family members, especially my parents whose encouragement and sharing enabled me to complete my studies. Without my family's support, I would not have finished this thesis.

Contents

Abstract **i**

Acknowledgements **iv**

1 Introduction **1**

1.1 Polynomial Constraint Solving 1

1.2 Existing Approaches 2

1.3 The Proposed Approach and Contributions 4

1.4 Thesis Outline 5

2 Abstract DPLL **7**

2.1 The Abstract DPLL Procedure 7

2.2 Abstract DPLL Modulo Theories 10

3 Abstract DPLL for Approximation Theories and Refinements **13**

3.1 Approximation Theories 13

3.2 DPLL(T) Procedure with Over and Under Approximation Theories 14

3.3 DPLL(T) Procedure with Refinement and Heuristics 16

3.4 Soundness and Completeness of Approximation Theories on Polynomial Inequality 17

3.4.1 Polynomial Inequality Constraints 17

3.4.2 Open Boxes as Topological Basis 18

3.4.3 Soundness and Completeness 19

4 Over and Under Approximations for Intervals **23**

4.1 Interval Arithmetic 23

4.1.1 Classical Interval 23

4.1.2 Affine Interval 25

4.1.3 Chebyshev Approximation Interval 32

4.2 Over and Under Approximations for Intervals 37

4.2.1 Interval Arithmetic as Over Approximation 37
4.2.2 Testing as Under Approximation 39

5 Strategies for Over/Under Approximations and Refinement 42
5.1 Strategies for Over and Under Approximations on Intervals 42
5.1.1 UNSAT Core in a Polynomial Inequality 42
5.1.2 Incremental Test Data Generation 45
5.2 Strategies for Refinement 48
5.2.1 Selecting Intervals to Decompose 48
5.2.2 Interval Decomposition 49

6 The SMT Solver raSAT and Experiments 56
6.1 Design Framework of raSAT 56
6.2 Experiments 60
6.2.1 Experiments on Different Measures 61
6.2.2 Experiments on Benchmarks of SMT-LIB 67

7 Extensions to Polynomial Equality 71
7.1 Greater-Than-or-Equal Handling 71
7.2 Polynomial Equality Handling 72
7.2.1 Polynomial Equality by Intermediate Value Theorem 72
7.2.2 Extension to Multiple Equalities 74

8 Conclusions 76
8.1 Summary of the Thesis 76
8.2 Future Directions 78
8.2.1 raSAT Development 78
8.2.2 Extensions of raSAT Loop 79
8.3 Applications 80

Publications 84

List of Figures

3.1 Refinement loop on $O.T$, $U.T$, refinements, and heuristics 17
3.2 Proof of completeness 22
3.3 Kissing situation 22
3.4 Convergence situation 22

4.1 Chebyshev approximation 33
4.2 Chebyshev approximation of x^2 and $x|x|$ 33
4.3 Results of polynomial constraint by IA 38
4.4 Results of testing 39
4.5 Strategy for random generation of test data 40

5.1 Incremental test data generation 47

6.1 Framework of **raSAT** 57
6.2 Interval decompositions by **raSAT** for Example 6.1.1 58
6.3 The choices of r are far from the threshold $\sqrt[n]{\frac{1}{k}}$ 63
6.4 The choices of r are close to the threshold $\sqrt[n]{\frac{1}{k}}$ 63

7.1 Solving equality by intermediate value theorem 73
7.2 Solving multiple equalities 74

List of Tables

5.1 Experimental results with and without UNSAT core 45
5.2 Experimental results for selecting of intervals to decompose 48
5.3 Experimental results for different strategies of interval decomposition . . . 55

6.1 Experimental results for $\psi = x_1^n + x_2^n < 1 \wedge (x_1 - r)^n + (x_2 - r)^n < 1$ . . . 62
6.2 Experimental results for $\psi = \sum_{i=1}^{k} x_i^n < 1 \wedge \sum_{i=1}^{k} (x_i - r)^n < 1$ 65
6.3 Experimental results for $\psi = \psi_1 \wedge \psi_2$ 66
6.4 Experimental results for Hong, Zankl, and Meti-Tarski families 68

7.1 Experimental results for 15 equality problems of Zankl family 73

Chapter 1

Introduction

1.1 Polynomial Constraint Solving

Polynomial constraint solving is to find an instance that satisfies given polynomial inequality/equality. For instance,

$$\exists x, y. -y^2 + (x^2 - 1)y - 1 > 0 \wedge -x^2 - y^2 + 4 > 0$$

is such an example. This is an easy formula, but proving its satisfiability and showing a satisfiable instance (e.g., $x = 1.8$, $y = 0.9$) are not so easy.

Many problems in hardware/software verifications and analysis can be reduced to polynomial constraint solving.

- **Automated detection of roundoff and overflow errors**, which is such an application [23, 24] in software verification. For instance, consider DSP decoder like mpeg4. Usually, the decoder definition is given by a reference algorithm in the programming language C, which uses floating point number. In an embedded system, it is tempting to replace floating point into fixed point numbers. However, naive replacement would cause recognizable noise and locating such roundoff error source is not easy.

- **Automatic termination proving**, which is reduced to finding a suitable termination ordering [17]. There are lots of termination provers, e.g., T$_{\mathsf{T}}$T$_{2}$ [1], Aprove [2].

[1] `http://cl-informatik.uibk.ac.at/software/ttt2/`
[2] `http://aprove.informatik.rwth-aachen.de`

- **Loop invariant generation**. The use of Farkas's lemma is a popular approach in linear loop invariant generation [7]. Farkas's lemma uses products of matrices, and it requires solving polynomial constraints of degree 2. Non-linear loop invariant generation [28] and hybrid systems [29] require more complex polynomials.

- **Mechanical control design**. PID control is simple but widely used. Fujitsu used polynomial constraint solving to design PID control of HDD head movement [1].

Solving polynomial constraints on real numbers is decidable [31], though that on integers is undecidable (Hilbert's 10th problem). Quantifier elimination by cylindrical algebraic decomposition (QE-CAD) [6] is a well known technique, which is implemented in Mathematica, Maple/SyNRAC, Reduce/Redlog, QEPCAD, and recently nlSAT [14]. An obstacle is that QE-CAD is DEXPTIME with respect to the number of variables. In practice, it works fine up to 5 variables and lower degrees, but becomes rapidly harder. For instance, solving a polynomial constraint with 8 variables and degree 10 requires over 20 hours on a supercomputer.

1.2 Existing Approaches

Currently decision procedures for solving polynomial constraints are classified into one (or combinations) of five categories.

1. **QE-CAD**. RAHD [26] is based on the core computation of QE-CAD proposed by Tarski. It applies different versions of QE-CAD implementations such as QEPCAD-B, Reduce/Redlog. Because QE-CAD is DEXPTIME complexity in the number of variables, solving problems with a lot of variables seems to be a challenge for QE-CAD variants.

2. **Interval constraint propagation (ICP)**. ICP applies interval arithmetic as an over approximation for propagating conflict in a background theory. Many decision procedures are based on ICP such as RSOLVER [27] and iSAT [12], which apply classical interval (CI). To remove unsatisfiable elements, while RSOLVER develops a pruning algorithm, iSAT apply a tight interaction of SAT solver and eager theory propagation.

3. **Bit-blasting**. In this category, problems are reduced to SAT solving problems. Input formulas are bit-blasting to propositional formulas, which are then solved by a SAT solver. MiniSMT [33] applies bounded bit encoding to represent rational numbers and then extend representations for some fragments of real numbers. MiniSMT can show satisfiability quickly, but due to the bound on representation, it cannot conclude unsatisfiability. UCLID [5] represents an input formula by a bit-vector formula on a given finite width of bits. UCLID also applies both under and over approximations to refine each other. Reducing number of bits to represent an input formula is regarded an under approximation, and constructing an over-approximation formula by removing some clauses from the original formula. UCLID aims at finite-precision integer arithmetic then unsatisfiable problems cannot be detected when applying for real numbers.

4. **Linearization**. Several decision procedures apply linearization for polynomial constraints and then call an SMT solver for linear constraints to solver constraints obtained from linearization. Barcelogic [3] linearizes polynomial constraints by applying case analysis, which instantiates one of arguments in multiplication with finitely possible integers in a given-bounded range. Barcelogic applies for integer domain with finite input ranges. CORD [13] uses another technique for linearization, called CORDIC (COrdinate Rotation DIgital Computer). Both Barcelogic and CORD apply Yices as an SMT solver for solving linear constraints.

5. **Virtual substitution (VS)**. VS method [32] is adapted for SMT solving. In SMT-RAT toolbox [9, 10], combinations of VS method and incremental fashion for SMT, less lazy and eager theory propagation, are implemented. Due to restriction in degree of variables, required degree 2 (or at most degree 4), solving constraints with higher degrees seems a challenge. Z3 [19], the winner in the category QF_NRA (Quantifier Free of Nonlinear Real Arithmetic) of SMT competition in 2011, also applies VS in combinations with ICP and decision procedures for linear arithmetic.

1.3 The Proposed Approach and Contributions

Our aims are to propose *an approach* and develop an *SMT solver* for *solving polynomial constraints*. First, the target problem of the thesis is *solving polynomial inequality constraints* coming from following reasons.

(a) In constructive analysis, solving equality constraints on real numbers is in general undecidable (decidable only for algebraic numbers), whereas solving inequality is decidable. In other words, $a > b$ is computable, whereas $a = b$ is not computable.

(b) Inequality allows approximations. For instance, for a polynomial f, an over-approximation evaluates the range $O.range(f)$ of values of f as a superset of $range(f)$. An under-approximation is opposite, and evaluates the range $U.range(f)$ as a subset of $range(f)$. Thus, if $O.range(f)$ stays in negative values, $f > 0$ is detected UNSAT, and if $U.range(f)$ contains a positive value, $f > 0$ is detected SAT.

(c) Solving polynomial inequality on real numbers is reduced to that on rational numbers. For instance, for a polynomial f, $f(x) > 0$ is satisfied with a real number x, it is possible to take an enough close rational number y to x such that $f(y) > 0$. The reduction to rational numbers also avoids roundoff-errors in implementations. Real number representation is typically by floating point numbers in practice, in which roundoff errors are not clear. Instead, rational numbers allow precise implementation, e.g., the numerical packages of Ocaml.

Our approach and contributions in the thesis are summarized as follows:

(i) We propose an approach of *iterative approximation refinement* for solving constraints, which is formalized as an *abstract DPLL(T) procedure* for *over/under-approximations* and *refinements* under a background theory T. An under approximation is sound for proving in the background theory T, and an over approximation is sound for disproving. When they neither prove nor disprove, *refinements* are applied to decompose an atomic formula of the input formula, i.e., ψ to $\psi_1 \vee \psi_2$ such that $\psi \Leftrightarrow \psi_1 \vee \psi_2$. The proposed approach combined DPLL(T) procedure with over/under-approximations and refinements is sound and complete for solving polynomial inequality constraints under certain restrictions.

(ii) We instantiate *interval arithmetic* to over approximation and *testing* to under approximation. A new form of affine interval, called *Chebyshev Affine Interval*, is proposed. Chebyshev Affine Interval has an advantage over current *affine intervals* such that it can keep sources of computation for high degree variables, which would be useful for guiding refinements.

(iii) The proposed approach is implemented as the SMT solver **raSAT**, which applies *interval arithmetic* (over-approximation, aiming to decide unsatisfiability), *testing* (under-approximation, aiming to decide satisfiability), and *refinements* on interval decompositions.

(iv) We propose UNSAT cores of polynomial constraints that can improve efficiency in theory propagation of SMT. Computation of UNSAT cores in polynomial constraints allows inferring other unsatisfiable domain when a particular domain is detected as unsatisfiable. We propose an approach for incremental test data generation which would be useful when performing a large number of test data (i.e., a large number of variables).

(v) We propose strategies for refinements such that choices of intervals to decompose and methods to decompose an interval into smaller intervals. These strategies are guided from interval arithmetic, testing results, test data, and polynomials.

(vi) The proposed approach is also extended for *greater-than-or-equal* ($\geq$) constraints, i.e., $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > 0$ for proving satisfiability, and for proving unsatisfiability $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > -\delta_i$ for $\delta_i > 0$.

(vii) We propose a non-constructive method for solving polynomial constraints including *equalities* based on *intermediate value theorem*.

1.4 Thesis Outline

The structure of the thesis is organized as follows:

- Chapter 2 introduces preliminaries about the abstract DPLL procedure for SAT solving and abstract DPLL modulo theories for SMT solving under a background theory T, DPLL(T).

- Chapter 3 proposes the abstract DPLL(T) for over/under-approximation theories and refinement. Soundness and (restricted) completeness of the procedure are also given in this chapter.

- Chapter 4 presents *interval arithmetic* as over-approximation theory and *testing* as under-approximation theory. A new form of affine interval called CAI is newly proposed.

- Chapter 5 proposes UNSAT cores of a polynomial inequality, which allow inferring domain of unsatisfiability, and an approach of incremental test data generation for performing large test data. Strategies for refinement, such as choices of intervals for decomposition and how to decompose an interval into smaller intervals, are also presented in Chapter 5.

- Chapter 6 shows the design framework of the SMT solver **raSAT** and demonstrates how **raSAT** works in an example by different strategies for interval decomposition. Experimental results of **raSAT** for preliminary evaluation and benchmarks of SMT-LIB are shown in the chapter.

- We extend our approach for greater-than-or-equal constraints and equality handling in Chapter 7.

- Finally, conclusions, future directions, and applications are given in Chapter 8.

Chapter 2

Abstract DPLL

2.1 The Abstract DPLL Procedure

In this section, we introduce the DPLL procedure [25] applied for searching a satisfying truth assignment for a given *conjunctive normal form* (CNF) formula F of propositional logic. The assignment is incrementally built step by step. At each step, a next assignment is deduced from a current assignment and the CNF formula F, which is called *boolean constraint propagation*, or by a non-deterministic guess (*decision*) on the truth value of one of the remaining undefined variables. If the search fail, it causes backtrack from wrong decisions.

As notational convention, for a finite set of atoms (propositional symbols) A, an atom $a \in A$ is a *positive* literal and $\neg a$ is a *negative* literal. The negation of a literal l, written $\neg l$, denotes $\neg a$ if l is a, and a if l is $\neg a$. A *clause* is denoted C which is a set of literals, and a *CNF* formula F is a set of clauses (it is also regarded as conjunctions of clauses $F = C_1 \wedge \cdots \wedge C_n$). M is a (partial truth) *assignment*, which is sequences of literals (it is also regarded as a set of literals), such that $\{a, \neg a\} \subseteq M$ for no a. If $l \in M$ then l is *true* in M, if $\neg l \in M$ then l is *false* in M, and *undefined* otherwise. M is a *full* assignment if no literal of F is undefined. The *empty* assignment is denoted $\emptyset$. A clause C is *true* in M if $C \cap M \neq \emptyset$, denoted $M \models C$, is *false* in M, denoted $M \models \neg C$, if all its literals are false in M, and is *undefined* otherwise. We write $M \models F$, if all clauses of F are true in M, which is called a *model* (a propositional model) of F. If F has no model then it is *unsatisfiable*. We write $F \models C$ if the clause C is true in all models of F. We denote $C \vee l$

for the clause including the literal l and all literals of C.

A *binary relation* over *states* is denoted $\Longrightarrow$, called the *transition relation*, where a state is either *fail* or a pair of an assignment M and a CNF F, denoted $M \| F$. The DPLL procedure consists of rules describing transitions from a state to another state.

Followings are four basic DPLL procedure rules,

- **UnitPropagate:**

$$M \parallel F \wedge (C \vee l) \Longrightarrow Ml \parallel F \wedge (C \vee l) \quad \textit{if} \quad \begin{cases} M \models \neg C \text{ and} \\ l \text{ is undefined in } M \end{cases}$$

- **Decide:**

$$M \parallel F \Longrightarrow Ml^d \parallel F \quad \textit{if} \quad \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \text{ and} \\ l \text{ is undefined in } M \end{cases}$$

  If l is selected for the decide rule, it is called a *decision literal*, denoted as l^d, in the assignment Ml^d.

- **Fail:**

$$M \parallel F \wedge C \Longrightarrow \textit{fail} \quad \textit{if} \quad \begin{cases} M \models \neg C \text{ and} \\ M \text{ contains no decision literals} \end{cases}$$

- **Backjump:**

$$Ml^d M_1 \parallel F \Longrightarrow Ml' \parallel F \quad \textit{if} \quad \begin{cases} \text{there is some clause } C \vee l' \text{ such that} \\ F \models C \vee l', \\ M \models \neg C, \\ l' \text{ is undefined in } M \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in a clause of } F \end{cases}$$

The DPLL procedure takes an input CNF formula F and computes a finite sequences of the form $\emptyset \parallel F \Longrightarrow \cdots \Longrightarrow S$. It starts from the initial state $\emptyset \parallel F$ and terminates when

it reaches to a final state S, which is either *fail* when F is unsatisfiable, or in the form of $M \parallel F'$, where M is a model of F ($M \models F$), when F is satisfiable. Note that the CNF formula F' in a final state can be different from the input F by adding some clauses during the DPLL procedure.

The *unitPropagate* rule (*unit propagation*) is applied for deducing a next assignment from the current assignment M and the current CNF formula F. *Decide* rule is applied for a non-deterministic guess on the truth value of an undefined literal. The *fail* and *backjump* rules are applied when there is a *conflict*, i.e., a clause in the current CNF formula F that is false in the current assignment M. If the CNF formula F is unsatisfiable, the *fail* rule is applied when there are no decision literals in M. Otherwise the *backjump* rule is applied.

Example 2.1.1. This is an example applying the basic DPLL rules for finding a model of the input CNF formula $F = (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5)$.

$$
\begin{array}{rll}
\emptyset & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \Longrightarrow \quad (Decide) \\
l_2^d & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \Longrightarrow \quad (UnitPropagate) \\
l_2^d\ l_1 & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \Longrightarrow \quad (Decide) \\
l_2^d\ l_1\ l_4^d & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \Longrightarrow \quad (UnitPropagate) \\
l_2^d\ l_1\ l_4^d \neg l_3 & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \Longrightarrow \quad (UnitPropagate) \\
l_2^d\ l_1\ l_4^d\ \neg l_3\ l_5 & \parallel (l_1 \vee \neg l_2) \wedge (\neg l_4 \vee \neg l_3) \wedge (\neg l_4 \vee \neg l_1 \vee l_5) & \quad \text{Final state: model found}
\end{array}
$$

In addition to basic rules, the DPLL procedure with clause learning consists of two additional rules, which are *learn* and *forget* rules.

- **Learn:**

$$M \parallel F \Longrightarrow M \parallel F \wedge C \quad \textit{if} \quad \begin{cases} F \models C \text{ and} \\ \text{all atoms of } C \text{ occur in } F \end{cases}$$

- **Forget:**

$$M \parallel F \wedge C \Longrightarrow M \parallel F \quad \textit{if} \quad F \models C$$

In these two rules, the clause C is said to be learned and forgotten, respectively. In

the *backjump* rule, the clause $C \vee l'$, called a *conflict clause*, is discovered by *implication graph* [11], which is applied for finding causes of a conflict. Then the conflict clause is learned by *learn* rule to avoid producing the same conflict. *Forget* rule is applied for free memory by removing a clause C with low *activity* (i.e., the number of times C causes conflict or unit propagation) [22].

Example 2.1.2. This is an example applying the *backjump* rule and the *learn* rule for finding a model of the input CNF formula $F = (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4)$.

$$
\begin{array}{rlll}
\emptyset & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (Decide) \\
l_1^d & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (UnitPropagate) \\
l_1^d\ l_2 & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (Decide) \\
l_1^d\ l_2\ l_3^d & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (UnitPropagate) \\
l_1^d\ l_2\ l_3^d\ l_4 & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (Backjump) \\
l_1^d\ l_2 & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) & \Longrightarrow & (Learn) \\
l_1^d\ l_2 & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) \wedge (\neg l_1 \vee \neg l_3) & \Longrightarrow & (UnitPropagate) \\
l_1^d\ l_2\ \neg l_3 & \| (\neg l_1 \vee l_2) \wedge (\neg l_3 \vee \neg l_2 \vee l_4) \wedge (\neg l_3 \vee \neg l_4) \wedge (\neg l_1 \vee \neg l_3) & & \text{Final state: model found}
\end{array}
$$

2.2 Abstract DPLL Modulo Theories

Satisfiability Modulo Theories (SMT) is a problem to detect satisfiable instances under a background theory. Whereas SAT solving, focusing only on satisfiability (SAT) of propositional formulas, SMT aims to detect satisfiable instances in more expressive logics. SMT separates case analysis as SAT solving and a decision procedure for a background theory T, denoted as DP_T, which is applied for checking *consistency* of atoms given by SAT solving, or deducing consequent from these atoms.

For example, a formula in the theory of non-linear arithmetic is,

$$F = (x > 0 \vee x < -2) \wedge (y > 0) \wedge (x^2 + y < 4)$$

Atoms of F are $x > 0$, $x < -2$, $y > 0$, and $x^2 + y < 4$. If SAT solving gives a model for F as $M = (x < -2), (y > 0), (x^2 + y < 4)$, then conjunction of literals in M is checked by

a DP_T. In this case, the DP_T decides satisfiability of $(x < -2) \wedge (y > 0) \wedge (x^2 + y < 4)$. If it is satisfiable, it is *T-consistent*, otherwise it is *T-inconsistent*. In this example, $(x < -2) \wedge (y > 0) \wedge (x^2 + y < 4)$ is T-inconsistent and $(x > 0) \wedge (y > 0) \wedge (x^2 + y < 4)$ is T-consistent.

We write $F \models_T \neg G$ if $F \wedge G$ is T-inconsistent.

Interaction between SAT solving and theory has *very lazy theory learning*, *less lazy theory learning*, and *eager theory propagation*, which are described below as *abstract DPLL modulo theories* [25].

- **Very lazy theory learning** interacts with DP_T when a full assignment is obtained from a SAT solver. If the theory DP_T disproves the full assignment, the SAT solver learns a clause $\neg l_1 \vee \cdots \vee \neg l_n \vee \neg l$ and is started again.

$$M l M_1 \parallel F \Longrightarrow \emptyset \parallel F \wedge (\neg l_1 \vee \cdots \vee \neg l_n \vee \neg l) \quad \textit{if} \quad \begin{cases} M l M_1 \models F, \\ \{l_1, \ldots, l_n\} \subseteq M, \text{ and} \\ l_1 \wedge \cdots \wedge l_n \models_T \neg l. \end{cases}$$

  Note that $M l M_1$ is a full assignment.

- **Less lazy theory learning** interacts with DP_T when a (partial) assignment is obtained from a SAT solver and the DP_T refutes $l_1 \wedge ... \wedge l_n \wedge l$, it will learns the clause $\neg l_1 \vee \cdots \vee \neg l_n \vee \neg l$.

$$M l M_1 \parallel F \implies M l M_1 \parallel F \wedge (\neg l_1 \vee \cdots \vee \neg l_n \vee \neg l) \quad \textit{if} \quad \begin{cases} \{l_1, \ldots, l_n\} \subseteq M, \\ l_1 \wedge \cdots \wedge l_n \models_T \neg l, \text{ and} \\ \neg l_1 \vee \cdots \vee \neg l_n \vee \neg l \notin F. \end{cases}$$

  Note that $M l M_1$ is possibly a partial assignment and the SAT solver does not need to restart when it learns a clause.

- **Eager theory propagation** interacts with DP_T during the DPLL procedure of SAT solving, and the DPLL procedure continues when the theory admits the current decisions. By applying this rule, next assignments are deduced from the current

assignment M based on the DP_T for the background theory T.

$$M \parallel F \Longrightarrow Ml \parallel F \quad \textit{if} \quad \begin{cases} M \models_T l, \\ l \text{ is undefined in } M, \text{ and} \\ l \text{ or } \neg l \text{ occurs in } F. \end{cases}$$

It is easy to separate work for SAT solving and the DP_T procedure by applying *very lazy theory learning*, thus we do not need to look inside process of SAT solving. Whereas *less lazy theory learning* and *eager theory propagation* require tighter interaction between SAT solving and DP_T, which needs internal modification on process of SAT solving.

Chapter 3

Abstract DPLL for Approximation Theories and Refinements

In this chapter, we introduce abstract DPLL(T) for sandwiching by over/under-approximation theories and their refinements. The DPLL(T) procedure applies over and under approximations for proving or disproving (i.e., *T-consistent*, *T-inconsistent* in Chapter 2) in the background theory T, respectively. When they neither prove nor disprove, refinement in the DPLL(T) procedure is applied, which leads to better approximations. Soundness and (restricted) completeness of the DPLL(T) procedure for approximations and refinements are also given in this chapter.

3.1 Approximation Theories

We start with a general framework, and assume that a target constraint F is a CNF formula of first-order logic. In Section 3.4.1, we will instantiate F as a conjunction of I and P, where I is an interval constraint, $x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)$, and P is a conjunction of polynomial inequalities, $\bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0$.

In the *very lazy theory learning*, full truth assignments (obtained from a SAT solver) are proved or disproved by a decision procedure (DP_T) in a background theory T. We abuse the symbol $\models_T$ in the following way. As notational convention, m (the lower case) denotes an instance (m is aimed at variable assignments) of variables appearing in literals, and M (the upper case) denotes a (full) truth assignment on literals. We regard M as a

conjunction of literals in a background theory T.

- If an instance m of variables appearing in F satisfies F, we denote $m \models_T F$.
- For a truth assignment M, if an instance m satisfies M we denote $m \in M$. If m satisfies F for each instance $m \in M$, we denote $M \models_T F$.

Definition 3.1.1. For a constraint F and a truth assignment M, we say that F is

- T-valid under M if $M \models_T F$,
- T-satisfiable (T-SAT) under M if $m \models_T F$ for some $m \in M$, and
- T-unsatisfiable (T-UNSAT) under M if $M \models_T \neg F$.

If T is clear from the context, we simply say valid, satisfiable, and unsatisfiable.

Definition 3.1.2. Let T, $O.T$, and $U.T$ be theories. We say that,

- $O.T$ is an *over-approximation theory* of T if $O.T$-UNSAT implies T-UNSAT, and
- $U.T$ is an *under-approximation theory* of T if $U.T$-SAT implies T-SAT.

We further assume that $O.T$-valid implies T-valid.

The intuition behind is that $O.T$ is applied for proving unsatisfiability (UNSAT) and $U.T$ is applied for proving satisfiability (SAT) of formulas in a background theory T. Later in Chapter 4, we will instantiate $O.T$ and $U.T$ with *interval arithmetic* and *testing*, respectively.

3.2 DPLL(T) Procedure with Over and Under Approximation Theories

When solving a constraint F by SMT, we first project each literal (in first-order logic) in F to a boolean variable (i.e., a literal of propositional logic), denoted by $proj(F)$, which is a CNF of propositional formula. SAT solver will give a truth assignment (if satisfiable).

We present the DPLL(T) procedure in very lazy theory learning, denoted as $\Longrightarrow_{VL}$, and use $\models_{O.T}$ and $\models_{U.T}$ for proving or disproving a full truth assignment given from SAT solver. It works as,

(i) M will be chosen by SAT solver and will be evaluated by $\models_{O.T}$ or $\models_{U.T}$.

(ii) If either $\models_{O.T}$ or $\models_{U.T}$ proves SAT (*SAT rule* is applied), the DPLL(T) procedure terminates and outputs SAT.

(iii) If $\models_{O.T}$ disproves (*very lazy theory learning rule* is applied), SAT solver will detect another M.

(iv) When $\models_{O.T}$ and $\models_{U.T}$ neither prove nor disprove, a *refinement rule* decomposes an atomic formula ψ in F to $\psi_1 \vee \psi_2$ such that they are mutually exclusive.

Assume that M is a conjunction of literals in F such that $proj(M) \models proj(F)$ (i.e., $proj(M)$ is a truth assignment returned by SAT solver).

- **SAT rule** is applied when either $M \models_{O.T} F$, or $m \models_{U.T} F$ for some $m \in M$.

$$M \parallel F \Longrightarrow_{VL} \ SAT \quad if \quad \begin{cases} M \models_{O.T} F, \text{ or} \\ m \models_{U.T} F \text{ for some } m \in M. \end{cases}$$

- **Fail rule** is applied when SAT solver returns UNSAT.

$$\emptyset \parallel F \Longrightarrow_{VL} \ fail \quad if \ proj(F) \text{ is UNSAT}$$

- **Very lazy theory learning rule** is applied when $M \models_{O.T} \neg F$.

$$M \parallel F \Longrightarrow_{VL} \emptyset \parallel (\neg l_1 \vee \cdots \vee \neg l_n) \wedge F \quad if \quad \begin{cases} \{l_1, \ldots, l_n\} \subseteq M \text{ and} \\ l_1 \wedge \cdots \wedge l_n \models_{O.T} \neg F. \end{cases}$$

Whenever either SAT rule or fail rule are applied, the DPLL(T) procedure terminates and informs SAT or UNSAT, respectively. Note that in *very lazy theory learning rule*, if $l_1 \wedge \cdots \wedge l_n$ is chosen to be minimal (on the number of literals), it is an *UNSAT core*, which causes unsatisfiability. Finding an UNSAT core improves efficiency in theory propagation (learning) of the $DPLL(T)$ procedure, which will be presented in Section 5.1.1.

3.3 DPLL(T) Procedure with Refinement and Heuristics

Definition 3.3.1. For a constraint F, ψ is an atomic formula in F if $prof(\psi)$ is a propositional literal of $prof(F)$.

When $\models_{O.T}$ and $\models_{U.T}$ neither prove nor disprove M, a *refinement rule* decomposes an atomic formula ψ in F to $\psi_1 \vee \psi_2$ such that theories are mutually exclusive ($\psi \Leftrightarrow \psi_1 \vee \psi_2$).

- **Refinement rule** is applied when $\models_{O.T}$ and $\models_{U.T}$ neither prove nor disprove M, and F is refined to $F' = (\psi \Leftrightarrow \psi_1 \vee \cdots \vee \psi_n) \wedge F$.

$$M \parallel F \Longrightarrow_{VL} \emptyset \parallel (\psi \Leftrightarrow \psi_1 \vee \cdots \vee \psi_n) \wedge F \quad \begin{array}{l} \textit{if } \psi \text{ is an atomic formula in } F \\ \text{and } \psi \Leftrightarrow \psi_1 \vee \cdots \vee \psi_n. \end{array}$$

An example of the refinement rule is described below for a target constraint F including constraints for variable ranges, which will be presented in Section 3.4.1 as interval constraints, i.e., $x \in (a, b)$ represented for $a < x < b$.

Example 3.3.2. Assume that $x \in (0, 2)$ is an atomic formula in F, it is refined into $x \in (0, 2) \Leftrightarrow x \in (0, 1] \vee x \in (1, 2)$.

$$M \parallel F \Longrightarrow_{VL} \emptyset \parallel (x \in (0, 2) \Leftrightarrow x \in (0, 1) \vee (x = 1) \vee x \in (1, 2)) \wedge F$$

A *heuristic rule* is to halt the DPLL(T) procedure by setting *termination heuristics* $isHalt$, i.e., refined too much, etc. In Section 3.4.3, we will give definition of $isHalt$ for polynomial inequality constraints. It is applied when the size of each interval becomes small enough, i.e., less than a given threshold.

- **Heuristic rule** learns a clause $\neg M$ when M holds $isHalt(M)$

$$M \parallel F \Longrightarrow_{VL} \emptyset \parallel \neg M \wedge F \quad \textit{if } isHalt(M)$$

If heuristic rules are applied in the DPLL(T) procedure, the fail rule becomes $\emptyset \parallel F \Longrightarrow_{VL} \textit{unknown}$ *if* $proj(F)$ is UNSAT, and the DPLL (T) procedure cannot con-

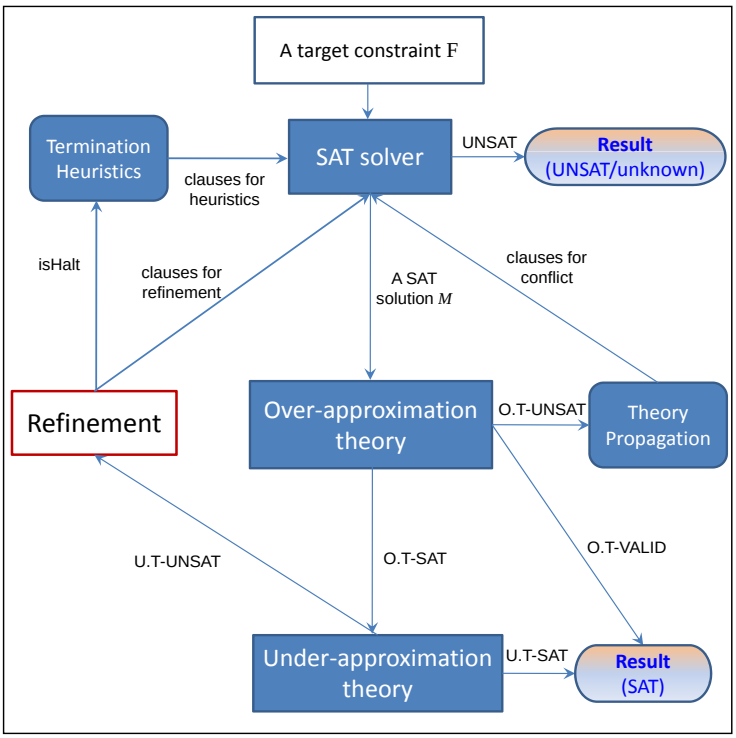

Figure 3.1: Refinement loop on $O.T$, $U.T$, refinements, and heuristics

clude UNSAT, though SAT rules correctly conclude SAT. Heuristic rule more focuses on detecting SAT, when the execution time becomes too long. Refinement loop of the DPLL(T) procedure that applies $O.T$, $U.T$, refinements, and heuristics is demonstrated in Figure 3.1.

3.4 Soundness and Completeness of Approximation Theories on Polynomial Inequality

3.4.1 Polynomial Inequality Constraints

We focus on *polynomial inequality constraints* with input ranges as open boxes which is described in Definition 3.4.1.

Definition 3.4.1. A polynomial inequality constraint $F = I \wedge P$ consists of

- an interval constraint $I = x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)$, and
- a polynomial constraint $P = \bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0$

for $a_i, b_i \in \mathbb{R}$ and a polynomial $f_i(x_1, \cdots, x_n)$ over variables $x_1, \cdots, x_n$. F is satisfiable if there is an instance satisfying $I \wedge P$. $f_i(x_1, \cdots, x_n) > 0$ is called an atomic polynomial inequality (API).

We also assume that all variables appearing in P appear in I. The intuition behind is that I and P describe input restrictions and a target property, respectively. By regarding I to be a truth assignment (i.e., $x_i \in (a_i, b_i)$ is regarded as a literal), we denote $I \models P$ if P is valid under I.

Example 3.4.2. An example of a polynomial inequality constraint with 2 variables and 2 APIs is,

$$F = x \in (-1, 3) \wedge y \in (2, 4) \wedge (x^3 y - y^4 > 0) \wedge (y^3 - xy > 0)$$

3.4.2 Open Boxes as Topological Basis

For mathematical simplicity, we prepare terminologies of topology [4, 16].

Definition 3.4.3. Let X be a set and $\mathbb{T}$ be a set of subsets on X. If $\mathbb{T}$ satisfies the following conditions:

(i) $\emptyset \in \mathbb{T}$ and $X \in \mathbb{T}$

(ii) If $U_1 \in \mathbb{T}$ and $U_2 \in \mathbb{T}$, then $U_1 \cap U_2 \in \mathbb{T}$

(iii) If $\mathbb{T}' \subset \mathbb{T}$ then $\cup \mathbb{T}' \in \mathbb{T}$,

$\mathbb{T}$ is a topology on X and X is called a topological space. A member of $\mathbb{T}$ is called an *open set* of X. A *closed set* of X is the complement of an open set. The intersection of all closed sets of X containing A is the *closure* of A, and the union of all open subsets of A is the *interior* of A.

Example 3.4.4. Let $\mathbb{T}$ be a set of all open intervals, $\{(a, b) \mid a, b \in \mathbb{R}, a \leq b\}$. $\mathbb{T}$ is a topology on $\mathbb{R}$ and the pair $(\mathbb{T}, \mathbb{R})$ is a topological space. An open interval (a, b) is an open set of $\mathbb{R}$ and a closed interval $[a, b]$ is a closed set of $\mathbb{R}$. The closure of A is the smallest closed set of X containing A, for instance, $closure((1, 4)) = [1, 4]$, $closure(\{(x, y) \mid x^2 - xy > 0\}) = \{(x, y) \mid x^2 - xy \geq 0\}$.

Definition 3.4.5. An *open box* of dimension n is a set $(a_1, b_1) \times \cdots \times (a_n, b_n)$ where $a_i, b_i \in \mathbb{R}, a_i \leq b_i$. For $\mathfrak{a} = (a_1, \cdots, a_n)$ and $\mathfrak{b} = (b_1, \cdots, b_n)$, we denote $(a_1, b_1) \times \cdots \times (a_n, b_n)$ by $(\mathfrak{a}, \mathfrak{b})$.

The set of all open boxes of dimension n is a topology on $\mathbb{R}^n$. We will consider covering by open boxes only, i.e., for a subset U of X, a covering is a set $\{B_\lambda\}$ of open boxes such that $U \subseteq \cup B_\lambda$. A set U is *compact*, if, for each covering of U, there exists its finite subset that is a covering of U. In Euclidian space, a set U is compact if, and only if, U is a bounded closed set.

Definition 3.4.6. Let X, Y be topological spaces and $f : X \mapsto Y$ be a map from X into Y. A map f is *continuous*, if, for each open set U of Y, $f^{-1}(U)$ is an open set of X.

3.4.3 Soundness and Completeness

In this section we discuss about soundness and (restricted) completeness of the DPLL(T) procedure, presented in Section 3.2 and 3.3, for polynomial inequality constraints defined in Definition 3.4.1.

Definition 3.4.7. Let F be polynomial inequality constraint, $\mathbb{S}(F) = \{x \in \mathbb{R}^n \mid F \text{ holds}\}$.

Since a polynomial is a continuous function, $\mathbb{S}(\bigwedge_{i=1}^{m} f_i > 0)$ is an open set. Since $\mathbb{Q}$ is dense in $\mathbb{R}$ (closure of $\mathbb{Q}$ is $\mathbb{R}$), next lemma is immediate.

Lemma 3.4.8. $\exists x \in \mathbb{R}^n.\ F(x) \Longleftrightarrow \exists x \in \mathbb{Q}^n.\ F(x)$

Lemma 3.4.8 says that proving SAT of F among real numbers is reduced to that among rational numbers.

Lemma 3.4.9. *Suppose that $a_j < b_j$ for $1 \leq j \leq n$ and f_i are polynomials. Assume $a_k < c < b_k$ for $1 \leq k \leq n$. Then, $x \in (a_1, b_1) \times \cdots \times (a_n, b_n) \wedge \bigwedge_{i=1}^{m} f_i > 0$ is SAT (resp. UNSAT) if, and only if, $(x \in (a_1, b_1) \times \cdots (a_k, c) \cdots \times (a_n, b_n) \vee x \in (a_1, b_1) \times \cdots (c, b_k) \cdots \times (a_n, b_n)) \wedge \bigwedge_{i=1}^{m} f_i > 0$ is SAT (resp. UNSAT).*

Proof. We show for the SAT case. If-part is obvious. For only-if-part, since $\mathbb{S}(\bigwedge_{i=1}^{m} f_i > 0)$ is an open set, if $y \in (a_1, b_1) \times \cdots \{c\} \cdots \times (a_n, b_n)$ satisfies $\bigwedge_{i=1}^{m} f_i > 0$, there exists

$x \in (a_1, b_1) \times \cdots (a_k, c) \cdots \times (a_n, b_n)$ (also $x \in (a_1, b_1) \times \cdots (c, b_k) \cdots \times (a_n, b_n)$) that satisfies $\bigwedge_{i=1}^{m} f_i > 0$. The same proof is applied for the UNSAT case.

For a polynomial inequality constraint $F = I \wedge P$, when an atomic formula $x \in (a, b)$ is refined into $x \in (a, b) \Leftrightarrow x \in (a, c) \vee x \in (c, b)$ (instead of $x \in (a, b) \Leftrightarrow x \in (a, c] \vee x \in (c, b)$), it does not change SAT (resp. UNSAT) from Lemma 3.4.9. We apply this interval decomposition as a refinement rule in abstract DPLL. Note that initially, I and P have conjunctions only. By refinements, I becomes a CNF, though P has conjunctions only. Thus, during abstract DPLL only $proj(I)$ is sent to a SAT solver, and then a full truth assignment M is in the form of $M = x_1 \in (l_1, h_1) \wedge \cdots \wedge x_n \in (l_n, h_n)$, which is a box represented for input ranges of variables. If P is a CNF, $prof(I \wedge P)$ is sent to a SAT solver, instead.

Let us recall Example 3.4.2 for a refinement rule.

Example 3.4.10. Let $M = I = x \in (-1, 3) \wedge y \in (2, 4)$, $x \in (-1, 3)$ and $y \in (2, 4)$ are refined to smaller intervals.

$$M \parallel F \Longrightarrow_{VL} \emptyset \parallel (x \in (-1, 3) \Leftrightarrow x \in (-1, 1) \vee x \in (1, 3)) \wedge (y \in (2, 4) \Leftrightarrow y \in (2, 3) \vee y \in (3, 4)) \wedge F$$

For a polynomial inequality constraint, termination condition $isHalt$ is defined based on length of intervals, i.e., less than a given threshold.

Definition 3.4.11. For $M = x_1 \in (l_1, h_1) \wedge \cdots \wedge x_n \in (l_n, h_n)$ and a bound $\delta > 0 \in \mathbb{R}$, $isHalt(M) = (h_1 - l_1 < \delta) \wedge \cdots \wedge (h_n - l_n < \delta)$.

For a polynomial inequality, we fix $DPLL(T)_{A.R}$ for the DPLL(T) procedure that applies the *SAT*, *Fail*, *Very Lazy Theory Learning*, *Refinement*, and $Heuristic$ rules. If we set the threshold for $isHalt$ enough small, we can conclude soundness and (restricted) completeness of $DPLL(T)_{A.R}$. Note that such threshold is not easy to compute (if we use QE-CAD algorithm, we may be able to compute), and in our **raSAT**, it is left as a heuristics.

Definition 3.4.12. Let $F = I \wedge P$ with $P = (\bigwedge_{i=1}^{m} f_i > 0)$ be a polynomial inequality constraint such that I is bounded. An over-approximation theory $O.T$ is *complete* (w.r.t.

F) if, for each $\delta > 0$ and $c = (c_1, \cdots, c_n)$ satisfying I, there exists $\gamma > 0$ such that $\bigwedge_{j=1}^{n} x_j \in (c_j - \gamma, c_j + \gamma) \models_{O.T} \bigwedge_{i=1}^{m} (f_i(c) - \delta < f_i(x) < f_i(c) + \delta)$.

Definition 3.4.13. Let $I = \bigwedge_{j=1}^{n} I_j$ for $I_j = x_j \in (a_j, b_j)$ and $F = I \wedge \bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0$. An interval decomposition strategy is *fair*, if, for each $c_j \in (a_j, b_j)$ and $\gamma > 0$, an interval decomposition for x_j for each j eventually occurs in $(c_j - \gamma, c_j + \gamma)$ (as long as neither $I' \models P$ nor $I' \models \neg P$, where I' is a decomposed box).

Theorem 3.4.14. *Let $I = \bigwedge_{j=1}^{n} I_j$ for $I_j = x_j \in (a_j, b_j)$, $F = I \wedge \bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0$ and $\mathbb{S}(f_i) = \{(x_1, \cdots, x_n) \mid f_i(x_1, \cdots, x_n) > 0\}$. Assume that an over-approximation theory $O.T$ is complete (w.r.t. F). If the threshold for $isHalt$ is enough small and an interval decomposition strategy is fair, the followings hold.*

- ***Soundness**: If $DPLL(T)_{A.R}$ reports SAT (UNSAT), F is really SAT (UNSAT).*
- ***Completeness**:*
  - *If F is SAT, $DPLL(T)_{A.R}$ eventually find SAT instances*
  - *If $\cap closure(\mathbb{S}(f_i)) = \emptyset$ and $closure(I)$ is compact, $DPLL(T)_{A.R}$ eventually detects UNSAT.*

Proof. *Soundness*: it is obvious from the definitions of O.T and U.T (the SAT rule).
Completeness: If F is SAT, $\cap\mathbb{S}(f_i) \neq \emptyset$, and there exists an open box in it with the size $\delta > 0$ (the left of Figure 3.2). If $\cap closure(\mathbb{S}(f_i)) = \emptyset$ and $closure(I)$ is compact, let $\delta(f_i)(x) = max\{|f_i(x) - f_1(x)|, \cdots, |f_i(x) - f_m(x)|\}$. Since $\cap closure(\mathbb{S}(f_i)) = \emptyset$, $\delta(f_i)(x) > 0$ for each i. Since $\delta(f_i)$ is continuous and $closure(I)$ is compact, $\delta(f_i)(x)$ has the minimal value for $x \in closure(I)$. Thus, $\delta_i = min\{\delta(f_i)(x) \mid x \in I\} > 0$. We set $\delta = \frac{min\{\delta_i\}}{2}$, and $\delta > 0$.

In either case, since $O.T$ is complete, there exists $\gamma > 0$ satisfying Definition 3.4.12. We set γ to be the threshold of $isHalt$. Since an interval decomposition strategy is fair, decomposed boxes detect either SAT or UNSAT, respectively.

Limitations for proving UNSAT come from kissing and convergence situations. Figure 3.3 describes an example of kissing situation for the constraint $x^2 + y^2 < 2^2 \wedge (x-4)^2 + (y-3)^2 < 3^2$. In this example, $closure(x^2 + y^2 < 2^2) \cap closure((x-4)^2 + (y-3)^2 < 3^2) = (x = 1.6, y = 1.2)$. Thus, the condition $\cap closure(\mathbb{S}(f_i)) = \emptyset$ avoids kissing situation.

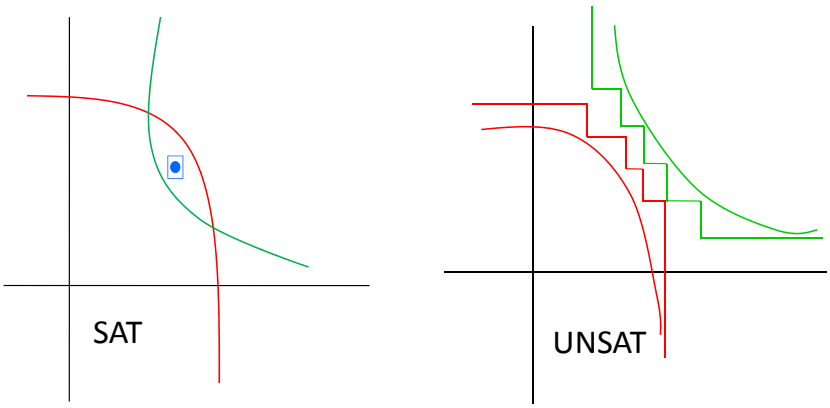

Figure 3.2: Proof of completeness

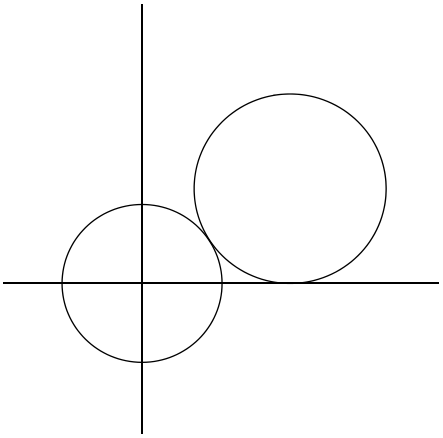

Figure 3.3: Kissing situation

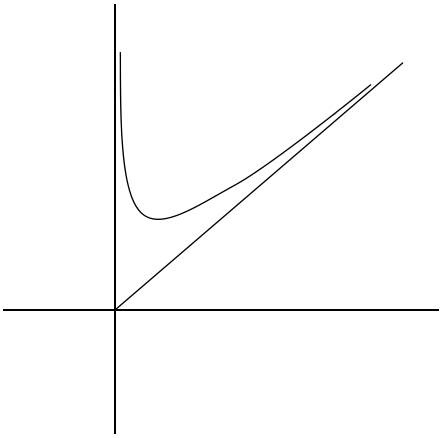

Figure 3.4: Convergence situation

Figure 3.4 is an example of convergence for the constraint $y > x + \frac{1}{x} \wedge y < x \wedge x > 0$. To avoid convergence, $closure(I)$ must be compact, i.e., bounded.

Note that the theorem requires only $O.T$ to be complete, since $O.T$-valid works as $U.T$-SAT. Later in Chapter 4, we apply an interval arithmetic as $O.T$ and testing as $U.T$. It is not difficult to see that an interval arithmetic is complete, and the aims of $U.T$ are,

- to obtain practical efficiency, and
- to guide interval decomposition (like "First Test-UNSAT" in Section 5.2.1).

Chapter 4

Over and Under Approximations for Intervals

We present *interval arithmetic* (IA) as an over-approximation theory, denoted as $\models_{O.T}$, and *testing* as an under-approximation theory, denoted as $\models_{U.T}$, in this chapter.

4.1 Interval Arithmetic

In this section, we first show interval arithmetic in the presentations of *classical interval* (CI) [21] and *affine arithmetic* (AF, AF_1, AF_2) [18], and we propose a new form of affine interval, called Chebyshev Affine Interval (CAI) [15].

4.1.1 Classical Interval

A popular example of IA is Classical Interval (CI), introduced in 1960s by Moore [21], which keeps a lower bound and an upper bound for representing a range of a variable.

Definition 4.1.1. CI arithmetic consisting of $\{+, -, \times, \div\}$ are defined as follows:

- $(a, b) + (c, d) = (a + c, b + d)$
- $(a, b) - (c, d) = (a - d, b - c)$
- $(a, b) \times (c, d) = (min(ac, ad, bc, bd), max(ac, ad, bc, bd))$
- $(a, b) \div (c, d) = (a, b) \times (\frac{1}{d}, \frac{1}{c})$ if $0 \notin (c, d)$

CI arithmetic is *over approximation* presented as the lemma below.

Lemma 4.1.2. *For $x \in (a, b)$ and $y \in (c, d)$, then $z = x \odot y \in (a, b) \odot (c, d)$ where $\odot \in \{+, -, \times, \div\}$.*

Proof. *It is obvious from the definitions of $\{+, -, \times, \div\}$ in CI arithmetic.*

Followings are examples of CI arithmetic.

Example 4.1.3. Let $x \in \bar{x} = (-2, 5)$ and $y \in \bar{y} = (4, 6)$. By using CI arithmetic, the bounds (*over-approximation bounds*) of $z = x \odot y$ ($\odot \in \{+, -, \times, \div\}$) are,

- addition $z = x + y$:

$$\begin{aligned} \bar{z} &= \bar{x} + \bar{y} \\ &= (-2, 5) + (4, 6) \\ &= (-2 + 4, 5 + 6) \\ &= (2, 11) \end{aligned}$$

  We can conclude that $z \in (2, 11)$. Note that $(2, 11)$ is an over-approximation bounds of z.

- subtraction $z = x - y$:

$$\begin{aligned} \bar{z} &= \bar{x} - \bar{y} \\ &= (-2, 5) - (4, 6) \\ &= (-2 - 6, 5 - 4) \\ &= (-8, 1) \end{aligned}$$

  then $z \in (-8, 1)$.

- multiplication $z = x \times y$:

$$\begin{aligned}
\bar{z} &= \bar{x} \times \bar{y} \\
&= (-2, 5) \times (4, 6) \\
&= (min(-2 \times 4, -2 \times 6, 5 \times 4, 5 \times 6), max(-2 \times 4, -2 \times 6, 5 \times 4, 5 \times 6)) \\
&= (min(-8, -12, 20, 30), max(-8, -12, 20, 30)) \\
&= (-12, 30)
\end{aligned}$$

  then $z \in (-12, 30)$.

- division $z = x \div y$:

$$\begin{aligned}
\bar{z} &= \bar{x} \div \bar{y} \\
&= (-2, 5) \div (4, 6) \\
&= (-2, 5) \times (\frac{1}{6}, \frac{1}{4}) \\
&= (min(-\frac{2}{6}, -\frac{2}{4}, \frac{5}{6}, \frac{5}{4}), max(-\frac{2}{6}, -\frac{2}{4}, \frac{5}{6}, \frac{5}{4})) \\
&= (-\frac{1}{2}, \frac{5}{4})
\end{aligned}$$

  We can conclude that

$$z \in (-\frac{1}{2}, \frac{5}{4}).$$

The weakness of CI is *loss of dependency* among values which leads imprecision for subtractions. For instance, if $x \in \bar{x} = (2, 4)$, then $x - x \in \bar{x} - \bar{x} = (2, 4) - (2, 4) = (-2, 2)$. To overcome the weakness of CI, Affine Interval is an alternative representation for interval arithmetic.

4.1.2 Affine Interval

Affine Interval (AI) [8, 18] introduces *noise symbols* ϵ, which are interpreted as values in $(-1, 1)$. AI allows to keep source of computation based on variable's noise symbols (ϵ) and it is likely to improve precision when applying subtractions among dependent values.

For instance, $x \in (2, 4)$ is represented as $x = 3 + \epsilon$, and $x - x = (3 + \epsilon) - (3 + \epsilon)$ is safely evaluated to 0.

Forms of AI vary by choices how to estimate multiplications. For instance, let $x \in (0, 2)$ and $y \in (1, 3)$, the affine form of x is $1 + \epsilon_1$ and the affine form of y is $2 + \epsilon_2$. Thus,

$$\begin{aligned} x^2 - x \times y &= (1 + \epsilon_1)^2 - (1 + \epsilon_1)(2 + \epsilon_2) \\ &= (1 + 2\epsilon_1 + \epsilon_1\epsilon_1) - (2 + \epsilon_2 + 2\epsilon_1 + \epsilon_1\epsilon_2) \\ &= -1 - \epsilon_2 + \epsilon_1\epsilon_1 - \epsilon_1\epsilon_2. \end{aligned}$$

Choices are,

(i) $\epsilon_1\epsilon_2$ is replaced with a fresh noise symbol (AF) [8, 30],

(ii) $\epsilon_1\epsilon_2$ is pushed into the fixed error noise symbol $\epsilon_\pm \in (-1, 1)$ (AF_1 and AF_2) [18],

(iii) $\epsilon_1\epsilon_2$ is replaced by $(-1, 1)\epsilon_1$ or $(-1, 1)\epsilon_2$ (EAI) [23],

(iv) $\epsilon_1\epsilon_1$ is replaced by the fixed positive noise symbol $\epsilon_+ \in (0, 1)$ or the negative noise symbol $\epsilon_- \in (-1, 0)$ (AF_2) [18] based on signs of coefficients.

Followings are the affine forms of AF, AF_1, AF_2 and their arithmetic.

The AF form

Definition 4.1.4. An AF of x is a formula of the form:

$$\ddot{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i$$

where $x \in (a_0 - \sum_{i=1}^{n} |a_i|, a_0 + \sum_{i=1}^{n} |a_i|)$ and $\epsilon_i \in (-1, 1)$ is a *noise symbol*.

For AF arithmetic, linear operations (i.e., addition and subtraction) are straightforward operations and nonlinear operations such that multiplication is applied (i) for approximating.

Definition 4.1.5. Let $\ddot{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i$ and $\ddot{y} = b_0 + \sum_{i=1}^{n} b_i\epsilon_i$. AF arithmetic consisting of $\{+, -, \times, \div\}$ are defined as follows:

- $\ddot{x} + \ddot{y} = (a_0 + b_0) + \sum_{i=1}^{n}(a_i + b_i)\epsilon_i$
- $\ddot{x} - \ddot{y} = (a_0 - b_0) + \sum_{i=1}^{n}(a_i - b_i)\epsilon_i$
- $\ddot{x} \times \ddot{y} = (a_0 b_0) + \sum_{i=1}^{n}(a_0 b_i + b_0 a_i)\epsilon_i + (\sum_{i=1}^{n}|a_i|)(\sum_{i=1}^{n}|b_i|)\epsilon_{n+1}$
- $\ddot{x} \div \ddot{y} = \ddot{x} \times \frac{1}{\ddot{y}} \quad$ if $0 \notin (b_0 - \sum_{i=1}^{n}|b_i|, b_0 + \sum_{i=1}^{n}|b_i|)$

where ϵ_{n+1} is a fresh noise symbol, interpreted as a value in $(-1, 1)$, and $\frac{1}{\ddot{y}}$ is computed by Chebyshev approximation [30].

Conversion between CI and AF

- *CI to AF*: for a CI $\bar{x} = (l, h)$, a corresponding AF form of x is $\ddot{x} = \frac{h+l}{2} + \frac{h-l}{2}\epsilon_x$.
- *AF to CF*: for a given AF $\ddot{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i$, the CI form of $\ddot{x}$ is $\bar{x} = (a_0 - \sum_{i=1}^{n}|a_i|, a_0 + \sum_{i=1}^{n}|a_i|)$.

Followings are examples of AF arithmetic.

Example 4.1.6. Let $x \in (1, 5)$ and $y \in (-1, 3)$. The AF forms of x and y are,

- $\ddot{x} = 3 + 2\epsilon_1$
- $\ddot{y} = 1 + 2\epsilon_2$

By using *AF* arithmetic, the bounds of $z = x \odot y$ ($\odot \in \{+, -, \times, \div\}$) are,

- addition $z = x + y$:

$$\begin{aligned}\ddot{z} &= \ddot{x} + \ddot{y} \\ &= 3 + 2\epsilon_1 + 1 + 2\epsilon_2 \\ &= 4 + 2\epsilon_1 + 2\epsilon_2\end{aligned}$$

The AF projection of $\ddot{z}$ is $(4 - 2 - 2, 4 + 2 + 2) = (0, 8)$. Then we can conclude that $z \in (0, 8)$.

- subtraction $z = x - y$:

$$\begin{aligned}\ddot{z} &= \ddot{x} - \ddot{y} \\ &= 3 + 2\epsilon_1 - 1 - 2\epsilon_2 \\ &= 2 + 2\epsilon_1 - 2\epsilon_2\end{aligned}$$

  The AF projection of $\ddot{z}$ is $(2 - 2 - 2, 2 + 2 + 2) = (-2, 6)$, then $z \in (-2, 6)$.

- multiplication $z = x \times y$:

$$\begin{aligned}\ddot{z} &= \ddot{x} \times \ddot{y} \\ &= (3 + 2\epsilon_1) \times (1 + 2\epsilon_2) \\ &= 3 + 6\epsilon_2 + 2\epsilon_1 + 4\epsilon_1\epsilon_2 \\ &= 3 + 4\epsilon_1 + 6\epsilon_2 + 4\epsilon_3\end{aligned}$$

  The AF projection of $\ddot{z}$ is $(3 - 4 - 6 - 4, 3 + 4 + 6 + 4) = (-11, 17)$. Then we can conclude that $z \in (-11, 17)$. Note that ϵ_3 is a new fresh noise symbol which is created by a multiplication from $\ddot{x} \times \ddot{y}$.

- division $z = x \div y$, we cannot compute the bounds of z because $0 \in (-1, 3)$.

The drawback of AF is increasing of fresh noise symbols when a number of non-linear operations is large.

The $\mathbf{AF_1}$ form

Definition 4.1.7. An AF_1 of x is a formula of the form:

$$\hat{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i + a_{n+1}\epsilon_{\pm}$$

where $x \in (a_0 - \sum_{i=1}^{n} |a_i| - a_{n+1}, a_0 + \sum_{i=1}^{n} |a_i| + a_{n+1})$, $\epsilon_i \in (-1, 1)$ is a *noise symbol*, $\epsilon_{\pm} \in (-1, 1)$ is the fixed error noise symbol and $a_{n+1} \geq 0$.

For AF_1 arithmetic, linear operations (i.e., addition and subtraction) are straightforward operations and nonlinear operations such that multiplication is applied (ii) for approximating.

Definition 4.1.8. Let $\hat{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i + a_{n+1}\epsilon_{\pm}$ and $\hat{y} = b_0 + \sum_{i=1}^{n} b_i\epsilon_i + b_{n+1}\epsilon_{\pm}$. AF_1 arithmetic consisting of $\{+, -, \times, \div\}$ are defined as follows:

- $\hat{x} + \hat{y} = (a_0 + b_0) + \sum_{i=1}^{n}(a_i + b_i)\epsilon_i + (a_{n+1} + b_{n+1})\epsilon_{\pm}$.
- $\hat{x} - \hat{y} = (a_0 - b_0) + \sum_{i=1}^{n}(a_i - b_i)\epsilon_i + (a_{n+1} + b_{n+1})\epsilon_{\pm}$.
- $\hat{x} \times \hat{y} = (a_0b_0) + \sum_{i=1}^{n}(a_0b_i + b_0a_i)\epsilon_i + (|a_0|b_{n+1} + |b_0|a_{n+1}) + (\sum_{i=1}^{n}|a_i|)(\sum_{i=1}^{n}|b_i|)\epsilon_{\pm}$
- $\hat{x} \div \hat{y} = \hat{x} \times \frac{1}{\hat{y}}$ if $0 \notin (b_0 - \sum_{i=1}^{n}|b_i| - b_{n+1}, b_0 + \sum_{i=1}^{n}|b_i| + b_{n+1})$

Note that $\frac{1}{\hat{y}}$ is computed by Chebyshev approximation [30], $a_{n+1}, b_{n+1} \geq 0$, and the coefficient of $\epsilon_{\pm}$ is $(a_{n+1} + b_{n+1})$ for subtraction $(-)$.

Followings are examples of AF_1 arithmetic.

Example 4.1.9. Recall from Example 4.1.6, the AF_1 form of x and y are,

- $\hat{x} = 3 + 2\epsilon_1$
- $\hat{y} = 1 + 2\epsilon_2$

By using AF_1 arithmetic, the bounds of $z = x \odot y$ $(\odot \in \{+, -, \times\})$ are,

- addition $z = x + y$ and subtraction $z = x - y$ are the same as AF.
- multiplication $z = x \times y$:

$$\begin{aligned}\hat{z} &= \hat{x} \times \hat{y} \\ &= (3 + 2\epsilon_1) \times (1 + 2\epsilon_2) \\ &= 3 + 6\epsilon_2 + 2\epsilon_1 + 4\epsilon_1\epsilon_2 \\ &= 3 + 4\epsilon_1 + 6\epsilon_2 + 4\epsilon_{\pm}\end{aligned}$$

The AF_1 projection of $\hat{z}$ is the same as AF ($z \in (-11, 17)$). Note that all of non-linear parts are pushed into $\epsilon_{\pm}$.

- division $z = x \div y$, we cannot compute because $0 \in (-1, 3)$.

AF_1 solves the problem of increasing fresh noise symbols in AF by approximating non-linear parts into the unique error noise symbol $\epsilon_{\pm}$.

The AF_2 form

Definition 4.1.10. An AF_2 of x is a formula of the form:

$$\breve{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i + a_{n+1}\epsilon_+ + a_{n+2}\epsilon_- + a_{n+3}\epsilon_{\pm}$$

where $x \in (a_0 - \sum_{i=1}^{n} |a_i| - a_{n+2} - a_{n+3}, a_0 + \sum_{i=1}^{n} |a_i| + a_{n+1} + a_{n+3})$, $\epsilon_i \in (-1, 1)$ is a *noise symbol*, $\epsilon_+ \in (0, 1)$ is the positive noise symbol, $\epsilon_- \in (-1, 0)$ is the negative noise symbol, $\epsilon_{\pm} \in (-1, 1)$ is the fixed error noise symbol and $a_{n+1}, a_{n+2}, a_{n+3} \geq 0$.

AF_2 applies (ii) and (iv) for approximating multiplications.

Definition 4.1.11. Let $\breve{x} = a_0 + \sum_{i=1}^{n} a_i\epsilon_i + a_{n+1}\epsilon_+ + a_{n+2}\epsilon_- + a_{n+3}\epsilon_{\pm}$ and $\breve{y} = b_0 + \sum_{i=1}^{n} b_i\epsilon_i + b_{n+1}\epsilon_+ + b_{n+2}\epsilon_- + b_{n+3}\epsilon_{\pm}$. AF_2 arithmetic consisting of $\{+, -, \times, \div\}$ are defined as follows:

- $\breve{x} + \breve{y} = (a_0 + b_0) + \sum_{i=1}^{n} (a_i + b_i)\epsilon_i + (a_{n+1} + b_{n+1})\epsilon_+ + (a_{n+2} + b_{n+2})\epsilon_- + (a_{n+3} + b_{n+3})\epsilon_{\pm}$.
- $\breve{x} - \breve{y} = (a_0 - b_0) + \sum_{i=1}^{n} (a_i - b_i)\epsilon_i + (a_{n+1} + b_{n+2})\epsilon_+ + (a_{n+2} + b_{n+1})\epsilon_- + (a_{n+3} + b_{n+3})\epsilon_{\pm}$.
- $\breve{x} \times \breve{y} = (a_0 b_0) + \sum_{i=1}^{n} (a_0 b_i + b_0 a_i)\epsilon_i + K_1\epsilon_+ + K_2\epsilon_- + K_3\epsilon_{\pm}$

  where:

  $$K_1 = \sum_{i=1, a_i b_i > 0}^{n+2} a_i b_i + \begin{cases} a_0 b_{n+1} + b_0 a_{n+1} & \text{if } a_0, b_0 \geq 0 \\ a_0 b_{n+1} - b_0 a_{n+2} & \text{if } a_0 > 0, b_0 < 0 \\ -a_0 b_{n+2} + b_0 a_{n+1} & \text{if } a_0 < 0, b_0 > 0 \\ -a_0 b_{n+2} - b_0 a_{n+2} & \text{if } a_0, b_0 < 0 \end{cases}$$

$$K_2 = \sum_{i=1, a_i b_i < 0}^{n+2} a_i b_i + \begin{cases} a_0 b_{n+2} + b_0 a_{n+2} & \text{if } a_0, b_0 \geq 0 \\ a_0 b_{n+2} - b_0 a_{n+1} & \text{if } a_0 > 0, b_0 < 0 \\ -a_0 b_{n+1} + b_0 a_{n+2} & \text{if } a_0 < 0, b_0 > 0 \\ -a_0 b_{n+1} - b_0 a_{n+1} & \text{if } a_0, b_0 < 0 \end{cases}$$

$$K_3 = \sum_{i=1}^{n+3} \sum_{j=1, j \neq i}^{n+3} |a_i b_j| + (|a_0| b_{n+3} + |b_0| a_{n+3}) + a_{n+3} b_{n+3}$$

- $\breve{x} \div \breve{y} = \breve{x} \times \frac{1}{\breve{y}}$ if $0 \notin (b_0 - \sum_{i=1}^{n} |b_i| - b_{n+1} - b_{n+2} - b_{n+3}, b_0 + \sum_{i=1}^{n} |b_i| + b_{n+1} + b_{n+2} + b_{n+3})$
  ($\frac{1}{\breve{y}}$ is computed by Chebyshev approximation [30])

Followings are examples of AF_2 arithmetic.

Example 4.1.12. Let $x \in (0, 2)$ and $y = 2 - x$. The AF_2 form of x and y are,

- $\breve{x} = 1 + \epsilon_1$
- $\breve{y} = 2 - (1 + \epsilon_1) = 1 - \epsilon_1$

By using AF_2 arithmetic, the bounds of $z = x \odot y$ ($\odot \in \{+, -, \times\}$) are,

- addition $z = x + y$:

$$\begin{aligned} \breve{z} &= \breve{x} + \breve{y} \\ &= (1 + \epsilon_1) + (1 - \epsilon_1) \\ &= 2 \end{aligned}$$

  We can conclude that $z = 2$.

- subtraction $z = x - y$:

$$\begin{aligned} \breve{z} &= \breve{x} - \breve{y} \\ &= (1 + \epsilon_1) - (1 - \epsilon_1) \\ &= 2\epsilon_1 \end{aligned}$$

  The AF_2 projection of $\breve{z}$ is $(-2, 2)$, then $z \in (-2, 2)$.

- multiplication $z = x \times y$:

$$\begin{aligned}\breve{z} &= \breve{x} \times \breve{y} \\ &= (1 + \epsilon_1) \times (1 - \epsilon_1) \\ &= 1 - \epsilon_1 + \epsilon_1 - \epsilon_1\epsilon_1 \\ &= 1 + \epsilon_-\end{aligned}$$

The AF_2 projection of $\breve{z}$ is $(0, 1)$ ($\epsilon_- \in (-1, 0)$), then $z \in$ **(0,1)**, which provides a better approximation than applying CI, AF and AF_1,

$CI: \quad z \in (0, 4)$

$AF: \quad z \in (0, 2)$

$AF_1: \quad z \in (0, 2)$

AF_2 can improve precision when approximating product of two same noise symbols, which are pushed into the positive or negative noise symbols (ϵ_+ or ϵ_-) depending on their coefficients.

4.1.3 Chebyshev Approximation Interval

In this section, we propose a new form of affine interval, called Chebyshev Approximation Interval (CAI), which is based on Chebyshev approximation.

Definition 4.1.13. A CAI of x is a formula of the form:

$$\mathring{x} = \bar{a}_0 + \sum_{i=1}^{n} \bar{a}_i\epsilon_i + \sum_{i=1}^{n} \bar{a}_{i+n}\epsilon_{i+n} + \bar{a}_{2n+1}\epsilon_{\pm}$$

where $\epsilon_i \in (-1, 1)$ is a noise symbol, $\epsilon_{\pm} \in (-1, 1)$ is the fixed error noise symbol, $\epsilon_{i+n} \in (0, 1)$ represents for the absolute value $|\epsilon_i|$ of ϵ_i, and a coefficient $\bar{a}_i$ represents for a CI.

Ideas behind are,

(v) introduction of noise symbols for absolute values ($\epsilon_{i+n} = |\epsilon_i|$) and

(vi) Chebyshev approximation of x^2 with noise symbols for absolute values.

(vi) comes from the observation that, for $x \in (-1, 1)$,

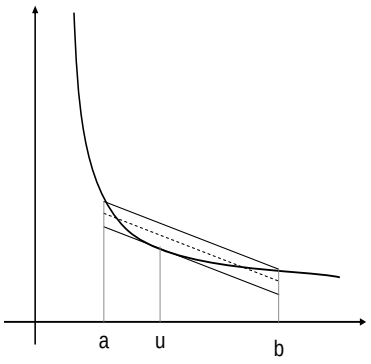

Figure 4.1: Chebyshev approximation

$$|x| - \tfrac{1}{4} \leq x^2 = |x|^2 < |x| \quad \text{and } x - \tfrac{1}{4} \leq x|x| \leq x + \tfrac{1}{4}$$

which are explained in Figure 4.2. This observation leads symbolic manipulation on products of the same noise symbol ϵ as

$$\epsilon\epsilon = |\epsilon||\epsilon| = |\epsilon| + (-\tfrac{1}{4}, 0) \quad \text{and} \quad \epsilon|\epsilon| = \epsilon + (-\tfrac{1}{4}, \tfrac{1}{4}).$$

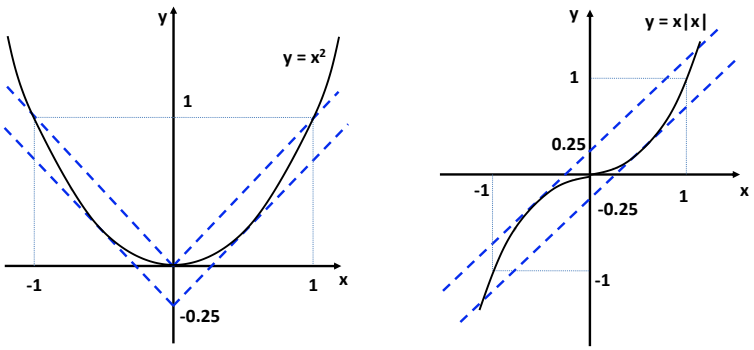

Figure 4.2: Chebyshev approximation of x^2 and $x|x|$

Definition 4.1.14. Let $\mathring{x}$ and $\mathring{y}$ be represented by CAI form,

- $\mathring{x} = \bar{a}_0 + \sum_{i=1}^{n} \bar{a}_i \epsilon_i + \sum_{i=1}^{n} \bar{a}_{i+n} \epsilon_{i+n} + \bar{a}_{2n+1} \epsilon_{\pm}$
- $\mathring{y} = \bar{b}_0 + \sum_{i=1}^{n} \bar{b}_i \epsilon_i + \sum_{i=1}^{n} \bar{b}_{i+n} \epsilon_{i+n} + \bar{b}_{2n+1} \epsilon_{\pm}$

and $\bar{c} = (-1, 1)$. CAI arithmetic consisting of $\{+, -, \times, \div\}$ are defined as follows *($\bar{a}\bar{b}$ is denoted for $\bar{a} \times \bar{b}$)*:

- $\mathring{x} + \mathring{y} = (\bar{a}_0 + \bar{b}_0) + \sum_{i=1}^{2n}(\bar{a}_i + \bar{b}_i)\epsilon_i + (\bar{c}\bar{a}_{2n+1} + \bar{c}\bar{b}_{2n+1})\epsilon_\pm$
- $\mathring{x} - \mathring{y} = (\bar{a}_0 - \bar{b}_0) + \sum_{i=1}^{2n}(\bar{a}_i - \bar{b}_i)\epsilon_i + (\bar{c}\bar{a}_{2n+1} + \bar{c}\bar{b}_{2n+1})\epsilon_\pm$
- $\mathring{x} \times \mathring{y} = K_0 + K_1\epsilon_i + K_2\epsilon_{i+n} + K\epsilon_\pm$, where $\{+, -, \times\}$ are CI arithmetic, and
  - $K_0 = \bar{a}_0\bar{b}_0 + \sum_{i=1}^{n}(\bar{a}_i\bar{b}_i(-\frac{1}{4}, 0) + \bar{a}_i\bar{b}_{i+n}(-\frac{1}{4}, \frac{1}{4}) + \bar{b}_i\bar{a}_{i+n}(-\frac{1}{4}, \frac{1}{4}) + \bar{a}_{i+n}\bar{b}_{i+n}(-\frac{1}{4}, 0))$
  - $K_1 = \sum_{i=1}^{n}(\bar{a}_0\bar{b}_i + \bar{a}_i\bar{b}_0 + \bar{a}_i\bar{b}_{i+n} + \bar{a}_{i+n}\bar{b}_i)$
  - $K_2 = \sum_{i=1}^{n}(\bar{a}_0\bar{b}_{i+n} + \bar{a}_{i+n}\bar{b}_0 + \bar{a}_i\bar{b}_i + \bar{a}_{i+n}\bar{b}_{i+n})$
  - $K = (\bar{c}\bar{a}_0\bar{b}_{2n+1} + \bar{c}\bar{b}_0\bar{a}_{2n+1}) + \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}\bar{c}\bar{a}_i\bar{b}_j + \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}\bar{c}\bar{a}_i\bar{b}_{j+n} + \sum_{i=1}^{n}\bar{c}\bar{a}_i\bar{b}_{2n+1} + \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}\bar{c}\bar{a}_{i+n}\bar{b}_j + \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n}\bar{c}\bar{a}_{i+n}\bar{b}_{j+n} + \sum_{i=1}^{n}\bar{c}\bar{a}_{i+n}\bar{b}_{2n+1} + \bar{c}\bar{a}_{2n+1}\bar{b}_{2n+1}$
- $\mathring{x} \div \mathring{y} = \mathring{x} \times \frac{1}{\mathring{y}}$ if $0 \notin$ projection of $\mathring{y}$ (CI bounds of $\mathring{y}$)

Note that $\epsilon_\pm$ is propagated from *unknown* sources, then its coefficient is computed by applying multiplication other coefficients with $\bar{c} = (-1, 1)$, and $\frac{1}{\mathring{y}}$ is computed by Chebyshev approximation [30].

Remark 4.1.15. Introduction of Chebyshev approximation is not new. For instance, Stolfi [30] proposed it based on the mean-value theorem, as in the Figure 4.1. Miyajima et al. [20] applied not only for products of the same noise symbols but also those of different noise symbols. However, their estimation on x^2 is only in the positive interval using the fact $x - \frac{1}{4} \leq x^2 < x$ for $x \in (0, 1)$. We newly introduce noise symbols for absolute values. The advantage is, coefficients are half compared to them, which reduce the effect of the offset $[-\frac{1}{4}, 0)$. Currently, we only focus on products of the same noise symbols, which is useful for computation of high degrees like in Taylor expansion.

Roughly speaking that CAI applies (ii), (v) and (vi). Followings are examples of CAI arithmetic.

Example 4.1.16. Let $f = x^3 - 3x + x^2$ with $x \in (-3, 1)$. The CAI form of x is $\mathring{x} = -1 + 2\epsilon$.

$$
\begin{aligned}
\mathring{f} &= \mathring{x}^3 - 3\mathring{x} + \mathring{x}^2 \\
&= (-1+2\epsilon) \times (-1+2\epsilon) \times (-1+2\epsilon) - (-3+6\epsilon) + (-1+2\epsilon) \times (-1+2\epsilon) \\
&= (1-4\epsilon+4\epsilon^2) \times (-1+2\epsilon) - (-3+6\epsilon) + (1-4\epsilon+4\epsilon^2) \\
&= (1-4\epsilon+4(|\epsilon| + (-\frac{1}{4}, 0))) \times (-1+2\epsilon) - (-3+6\epsilon) + (1-4\epsilon+4(|\epsilon| + (-\frac{1}{4}, 0))) \\
&= (1-4\epsilon+4|\epsilon| + (-1, 0)) \times (-1+2\epsilon) - (-3+6\epsilon) + (1-4\epsilon+4|\epsilon| + (-1, 0)) \\
&= ((0,1) - 4\epsilon + 4|\epsilon|) \times (-1+2\epsilon) - (-3+6\epsilon) + ((0,1) - 4\epsilon + 4|\epsilon|) \\
&= ((-1,0) + (0,2)\epsilon + 4\epsilon - 8\epsilon^2 - 4|\epsilon| + 8\epsilon|\epsilon|) - (-3+6\epsilon) + ((0,1) - 4\epsilon + 4|\epsilon|) \\
&= ((-1,0) + (0,2)\epsilon + 4\epsilon - 8(|\epsilon| + (-\frac{1}{4}, 0)) - 4|\epsilon| + 8(\epsilon + (-\frac{1}{4}, \frac{1}{4}))) - (-3+6\epsilon) + ((0,1) - 4\epsilon + 4|\epsilon|) \\
&= ((-1,0) + (0,2)\epsilon + 4\epsilon - 8|\epsilon| - (-2,0) - 4|\epsilon| + 8\epsilon + (-2,2)) - (-3+6\epsilon) + ((0,1) - 4\epsilon + 4|\epsilon|) \\
&= ((-3,4) + (12,14)\epsilon - 12\epsilon) - (-3+6\epsilon) + ((0,1) - 4\epsilon + 4|\epsilon|) \\
&= (0,8) + (2,4)\epsilon - 8|\epsilon|
\end{aligned}
$$

Note that we write a real number r for representation of a CI (r, r), i.e., 4 is represented for $(4, 4)$. For CAI projection, we apply *case analyses* for $|\epsilon|$, which are $\epsilon \in (0, 1)$ and $\epsilon \in (-1, 0)$,

- if $\epsilon \in (0, 1)$, then $\mathring{f} = (0, 8) + (2, 4)\epsilon - (8, 8)\epsilon = (0, 8) + (-6, -4)\epsilon$. The projection of $\mathring{f}$ is $(-6, 8)$.
- if $\epsilon \in (-1, 0)$, then $\mathring{f} = (0, 8) + (2, 4)\epsilon + (8, 8)\epsilon = (0, 8) + (10, 12)\epsilon$. The projection of $\mathring{f}$ is $(-12, 8)$.

For $\epsilon \in (-1, 1)$, the projection of $\mathring{f}$ is $(-6, 8) \cup (-12, 8) = (-12, 8)$, then $f \in$ **(-12,8)**. In comparison with CI, AF_1 and AF_2, we have the following results:

CI : $f \in (-33, 27)$

AF_1 : $f \in (-25, 31)$

AF_2 : $f \in (-13, 19)$

In Example 4.1.16, the affine form of x is $-1 + 2\epsilon$. When approximating for ϵ^3, AF_2 pushes it into the fixed error noise symbol ($\epsilon_\pm$) while CAI estimates it by a form of ϵ,

which allows to keep information about sources of computation. In comparison with AF_2, CAI can keep sources of computation for high degrees (i.e., degrees ≥ 3), whereas AF_2 can handle up to degree 2.

Like CI, AI's arithmetic is over approximation, which is presented as the lemma below.

Lemma 4.1.17. *For $x \in (a, b)$ and $y \in (c, d)$, $\mathcal{x}$ and $\mathcal{y}$ are denoted for the affine forms of x and y in AF, AF_1, AF_2, or CAI. Then $z = x \odot y \in \mathcal{x} \odot \mathcal{y}$ where $\odot \in \{+, -, \times, \div\}$.*

Proof. *Proofs are directly obtained from the definitions of AF, AF_1, AF_2, and CAI arithmetic for $\{+, -, \times, \div\}$.*

We present below two additional examples to compare the results of CAI with CI, AF_1 and AF_2. The first one is a polynomial of degrees 4 and the second one is a *Taylor expansion* of *sin(x)* function, which is expanded to degree 9. In both of them, CAI gives the best bounds.

Example 4.1.18. Given $f = (x^2 - 2y^2 + 7)^2 + (3x + y - 5)^2$ with $x \in (-1, 1)$ and $y \in (-2, 0)$, the bounds of f computed by CI, AF_1, AF_2 and CAI are,

CI : $(-12, 164)$

AF_1 : $(-98, 220)$

AF_2 : $(-53, 191)$

CAI : **(-4.6875,163.25)**

Example 4.1.19. Given $sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$ with $x \in (0, 0.523598)$ (x is ranged from 0 to $\frac{\pi}{6}$ and $\pi = 3.141588$), the bounds of $sin(x)$ computed by CI, AF_1, AF_2 and CAI are,

CI : $10^{-6}($ $-23926.630584,$ $523925.958917)$

AF_1 : $10^{-6}($ $-6290.490992,$ $523927.832027)$

AF_2 : $10^{-6}($ $-6188.005805,$ $514955.797111)$

$\mathbf{CAI}_1$: $10^{-6}($ $-1591.614677,$ $503782.471931)$

When IA has a noise symbol ϵ, we define *sensitivity* [23] of a variable as the absolute value of the coefficient of corresponding ϵ. CAI can keep information about sources of

computation for high degrees. For instance, in CAI form of Example 4.1.20, the coefficient **3** of $|\epsilon_{\mathbf{1}}|$ has the largest sensitivity, which indicates x is the most influential.

Example 4.1.20. Let $f = x^3 - 2xy$ with $x \in (0, 2)$ ($x = 1 + \epsilon_1$) and $y \in (1, 3)$ ($y = 2 + \epsilon_2$), we have,

- by AF_2, $\breve{f} = -3 - \epsilon_1 - 2\epsilon_2 + 3\epsilon_+ + 3\epsilon_\pm$ and the bounds of f are estimated as $(-9, 6)$,
- by CAI, $\mathring{f} = (-4, -\frac{11}{4}) + (-\frac{1}{4}, 0)\epsilon_1 - 2\epsilon_2 + \mathbf{3}|\epsilon_{\mathbf{1}}| + (-2, 2)\epsilon_\pm$ and the bounds of f are estimated as **(-8,4.5)**.

4.2 Over and Under Approximations for Intervals

4.2.1 Interval Arithmetic as Over Approximation

Interval arithmetic (IA) is applied for estimating bounds of polynomials under a given input range (a box), and we use it as an over-approximation theory. We instantiate IA to $O.T$ in Chapter 3, and obtain the definition below.

Definition 4.2.1. Given an interval constraint $I = x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)$, a polynomial constraint P of the form

$$\bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0.$$

Let $f_i^l(x_1, \cdots, x_n)$ and $f_i^u(x_1, \cdots, x_n)$ be lower and upper bounds estimated by IA when x_i holds the interval constraint I. We say,

- P is *IA-VALID* under I, if IA evaluates $\forall i \in (1, m).\ f_i^l(x_1, \cdots, x_n) > 0$,
- P is *IA-UNSAT* under I, $\exists i \in (1, m).\ f_i^u(x_1, \cdots, x_n) \leq 0$, and
- P is *IA-SAT* under I, if $(\exists j \in (1, m).\ f_j^l(x_1, \cdots, x_n) \leq 0) \wedge (\bigwedge_{i=1}^{m} f_i^u(x_1, \cdots, x_n) > 0)$.

Note that lower and upper bounds estimated by IA are over-approximation bounds (Lemma 4.1.2, 4.1.17). Figure 4.3 shows results of a polynomial constraint decided by IA. IA-VALID and IA-UNSAT safely reason satisfiability (SAT) and unsatisfiability (UNSAT), respectively. However, IA-SAT cannot conclude SAT. IA-SAT is regarded as *unknown* and shifted to under approximation (testing) for finding a SAT solution. For interval arithmetic, we aim at applying affine intervals such as AF, AF_1, AF_2, and CAI.

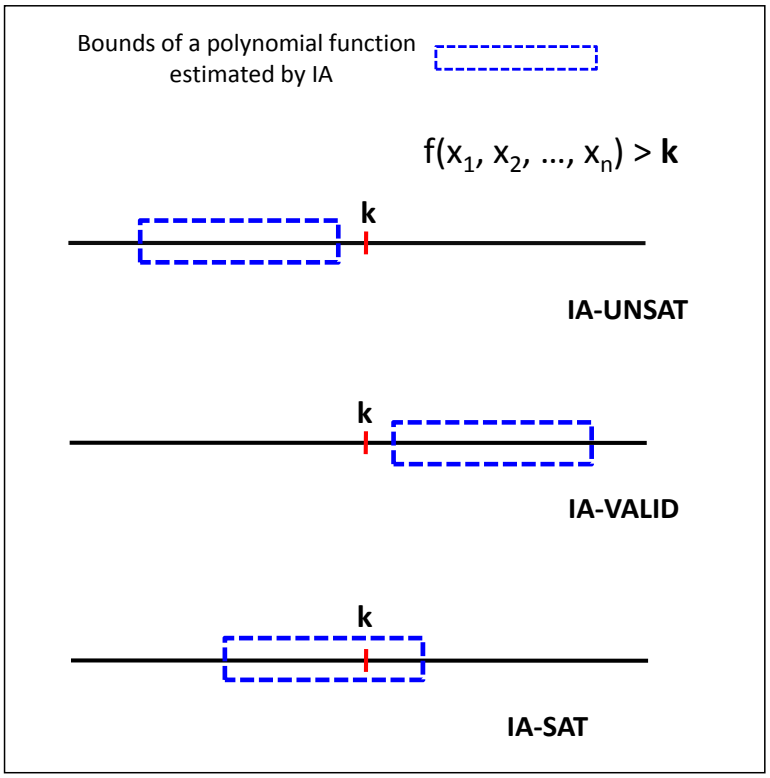

Figure 4.3: Results of polynomial constraint by IA

Example 4.2.2. Given an interval constraint $I = x \in (0, 2) \wedge y \in (1, 3)$ and three polynomial constraints,

- $(f_1 = x^5 - 3x^2y + y^3 + 20) > 0$
- $(f_2 = xy^2 - 2xy - y^3 - 7) > 0$
- $(f_3 = x^2y^3 - y^4) > 0$

by applying CAI arithmetic, the bounds of polynomials are

- $f_1 \in (0.9375, 67.75)$,
- $f_2 \in (-38.75, 0)$ and
- $f_3 \in (-131.5, 94.125)$

Then, we can conclude that

- $f_1 > 0$ is *IA-VALID* under I,
- $f_2 > 0$ is *IA-UNSAT* under I and
- $f_3 > 0$ is *IA-SAT* under I.

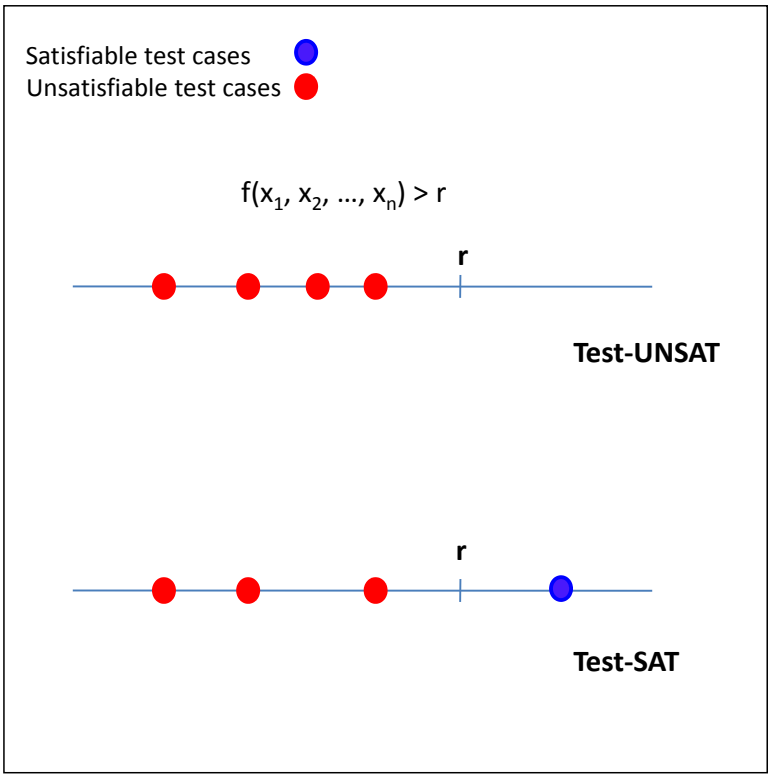

Figure 4.4: Results of testing

4.2.2 Testing as Under Approximation

We instantiate testing to $U.T$ in Chapter 3 and obtain the definition below.

Definition 4.2.3. Given an interval constraint $I = x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)$, a polynomial constraint P of the form

$$\bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0,$$

and a choice function $\theta : (\mathbb{R} \times \mathbb{R})^n \to \mathbb{R}^n$ with $\theta(I) \in (a_1, b_1) \times \cdots \times (a_n, b_n)$.

For a finite set Θ of choice functions, we say

- P is *Test-SAT* under I if P holds for some $\theta \in \Theta$, and
- P is *Test-UNSAT* under I if P never holds for each $\theta \in \Theta$.

The set Θ of choice functions in Definition 4.2.3 is a set of test data. Note that Test-SAT implies SAT and Test-UNSAT does not imply UNSAT (Figure 4.4), which will be applied for refinements (i.e., an interval is decomposed into smaller intervals).

There are two immediate strategies [24] to generate random test data.

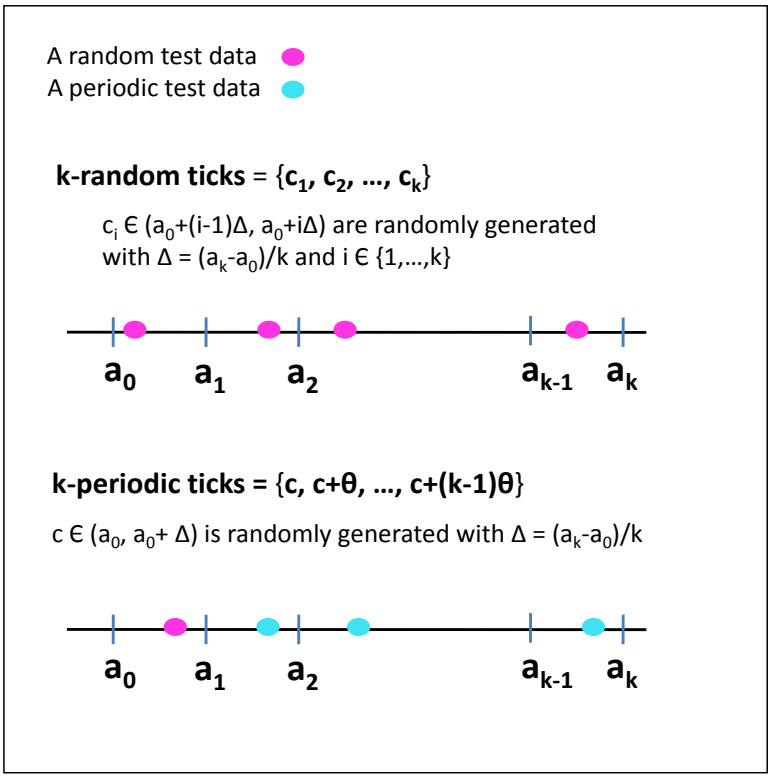

Figure 4.5: Strategy for random generation of test data

Definition 4.2.4. For an interval (l, h) and $k \geq 1$,

- the k-**random ticks** are $\{c_1, \cdots, c_k\}$, and
- the k-**periodic ticks** are $\{c, c + \Delta, \cdots, c + (k-1)\Delta\}$,

where $\Delta = \frac{h-l}{k}$, $c_i \in (l + (i-1)\Delta, l + i\Delta)$ and $c \in (l, l + \Delta)$ are randomly generated (with $i \in \{1, ..., k\}$).

Example 4.2.5. Recall from Example 4.2.2, for an interval constraint $I = x \in (0, 2) \wedge y \in (1, 3)$ and a polynomial constraint $(f_3 = x^2y^3 - y^4) > 0$, $f_3 > 0$ is shifted to testing because it is IA-SAT. By applying 2-random ticks for generating test data,

- if the test data in Θ_1 is $\{x = 0.5, x = 1.2, y = 1.5, y = 2.5\}$

  $x = 0.5, y = 1.5 \quad f_3 = x^2y^3 - y^4 = -4.21875$

  $x = 0.5, y = 2.5 \quad f_3 = x^2y^3 - y^4 = -35.15625$

  $x = 1.2, y = 1.5 \quad f_3 = x^2y^3 - y^4 = -0.2025$

  $x = 1.2, y = 2.5 \quad f_3 = x^2y^3 - y^4 = -16.5625$

then $f_3 > 0$ is Test-UNSAT in Θ_1.

- If the test data in Θ_2 is $\{x = 0.8, x = 1.7, y = 1.5, y = 2.5\}$

  $x = 0.8, y = 1.5 \quad f_3 = x^2y^3 - y^4 = -2.9025$

  $x = 0.8, y = 2.5 \quad f_3 = x^2y^3 - y^4 = -29.0625$

  $x = 1.5, y = 1.5 \quad f_3 = x^2y^3 - y^4 = \mathbf{2.53125}$

  $x = 1.5, y = 2.5 \quad f_3 = x^2y^3 - y^4 = -3.90625$

  then $f_3 > 0$ is Test-SAT in Θ_2 by the test data $x = 1.5$ and $y = 1.5$. We can conclude that $x \in (0, 2) \wedge y \in (1, 3) \wedge (f_3 = x^2y^3 - y^4) > 0$ is SAT.

Chapter 5

Strategies for Over/Under Approximations and Refinement

5.1 Strategies for Over and Under Approximations on Intervals

5.1.1 UNSAT Core in a Polynomial Inequality

The aim of (*very lazy*) *theory learning* is to remove UNSAT domain (UNSAT boxes) from searching domain. If we can infer other UNSAT boxes from a particular UNSAT box, it would be useful for theory propagation in the sense that a large UNSAT domain is removed from the searching domain.

Generally, an UNSAT core is defined as a minimal set $M_0 = \{l_1, \cdots, l_n\} \subseteq M$ (M is a full truth assignment given from a SAT solver represented for input ranges of variables) that disproves F (w.r.t. $\models_{O.T}$) in the very lazy theory learning rule (Section 3.2). To obtain a precise minimal M_0 is not easy. As a strategy to obtain smaller M_0, we introduce an UNSAT core $\hat{f}$ of a polynomial f based on the IA-UNSAT judgment. Then, M_0 is selected as literals in M corresponding to variables in $\hat{f}$.

Definition 5.1.1. $\hat{f}$ is an UNSAT core of a polynomial f if IA-UNSAT of $\hat{f} > 0$ implies IA-UNSAT of $f > 0$.

Definition 5.1.1 says that, for an interval constraint I, if $\hat{f} > 0$ is IA-UNSAT under I, then $f > 0$ is IA-UNSAT under I.

We apply the following steps for UNSAT core computation,

(a) finding an (API) $f > 0$ causing unsatisfiability of a given polynomial inequality when a conflict (IA-UNSAT) occurs,

(b) finding all UNSAT cores $\hat{f}$ of f, and

(c) generating learning clauses from UNSAT cores $\hat{f}$ by selecting only literals corresponding to variables of $\hat{f}$.

To compute an UNSAT core $\hat{f}$ of a polynomial f, we consider sub polynomials of f that cause unsatisfiability of the API $f > 0$. Ideas are,

- first, f is represented by two sub polynomials f_1 and f_2 from monomials of f such that $f = f_1 + f_2$,
- if $f_2 \leq 0$ for all values of its variables, then f_1 is an UNSAT core of f.

Example 5.1.2. For a polynomial inequality $(f = x^2 - xy - xz) > 0$ with $x, y, z \in (0, \infty)$, f consists of 2 UNSAT cores, which are,

- $\hat{f}_1 = x^2 - xy$, because $f = \hat{f}_1 - xz$ and $-xz < 0$ for $x, z \in (0, \infty)$
- $\hat{f}_2 = x^2 - xz$, because $f = \hat{f}_2 - xy$ and $-xy < 0$ for $x, y \in (0, \infty)$

Finding UNSAT cores improves efficiency in theory propagation (learning) of the DPLL(T) procedure. It only learns minimal clauses, clauses with minimal numbers of literals. The example below demonstrates uses of applying UNSAT cores for the very lazy theory learning rule.

Example 5.1.3. For a polynomial inequality (taken from *Hong* benchmark of SMT-LIB [2]) $P = (1 - x_0^2 - x_1^2 - \cdots - x_9^2 > 0) \wedge (x_0 x_1 \cdots x_9 - 1 > 0)$. Interval constraint is presented in the form of $x \in (-\infty, -1) \vee x \in (-1, 1) \vee x \in (1, \infty)$ for each variable x.

Considering a polynomial $f = 1 - x_0^2 - x_1^2 - \cdots - x_9^2$, UNSAT cores of f are,

$$
\begin{array}{llll}
1 - x_0^2 & 1 - x_0^2 - x_1^2 & 1 - x_0^2 - x_2^2 & 1 - x_0^2 - x_1^2 - x_2^2 \quad \cdots \\
1 - x_1^2 & 1 - x_1^2 - x_2^2 & 1 - x_1^2 - x_3^2 & 1 - x_1^2 - x_2^2 - x_3^2 \quad \cdots \\
1 - x_2^2 & 1 - x_2^2 - x_3^2 & 1 - x_2^2 - x_4^2 & 1 - x_2^2 - x_3^2 - x_4^2 \quad \cdots \\
1 - x_3^2 & 1 - x_3^2 - x_4^2 & 1 - x_3^2 - x_5^2 & 1 - x_3^2 - x_4^2 - x_5^2 \quad \cdots \\
\cdots & & & \\
1 - x_9^2 & & &
\end{array}
$$

Assume that $M = (x_0 \in (1, \infty)) \wedge \cdots \wedge (x_9 \in (1, \infty))$ is chosen from SAT solver. P is IA-UNSAT under M, then the very lazy learning rule learns the clause

$$\neg M = \neg(x_0 \in (1, \infty)) \vee \neg(x_1 \in (1, \infty)) \vee \cdots \vee \neg(x_9 \in (1, \infty))$$

as usual.

By applying UNSAT cores, the very lazy theory learning rule learns only clauses with minimal number of literals, which are,

$$
\begin{array}{lll}
\neg(x_0 \in (1, \infty)) & \text{from the UNSAT core} & 1 - x_0^2 \\
\neg(x_1 \in (1, \infty)) & \text{from the UNSAT core} & 1 - x_1^2 \\
\cdots & & \\
\neg(x_9 \in (1, \infty)) & \text{from the UNSAT core} & 1 - x_9^2
\end{array}
$$

These clauses help to remove large UNSAT boxes from the searching boxes. Note that we consider the UNSAT cores $1 - x_0^2, 1 - x_1^2, \cdots, 1 - x_9^2$ rather than others, i.e.,$1 - x_0^2 - x_1^2, 1 - x_0^2 - x_2^2, \cdots$ because their minimal number of variables lead to create short clauses for learning, which are more useful in theory propagation.

On the other hand, learning clauses from UNSAT cores avoids learning a large number of other clauses. For instance, learning the clause $\neg(x_0 \in (1, \infty))$ prevents 3^9 clauses from being learning as usual (i.e., for the remaining 9 variables and 3 choices $(-\infty, -1), (-1, 1), (1, \infty)$ for each, then their combinations are 3^9 clauses as total).

UNSAT core: on/off	SAT		UNSAT	
	No. Problems	Time(s)	No. Problems	Time(s)
With UNSAT core	37	**983.921**	9	0.512
Without UNSAT core	38	1480.812	9	0.512

Table 5.1: Experimental results with and without UNSAT core

We compare efficiency of **raSAT** when applying and not applying UNSAT core, and their results are shown in Table 5.1. The problems in the experiment are taken from 151 inequality problems of *Zankl* family (in the division QF_NRA of SMT-LIB [2]). Among them, 41 problems are solved with UNSAT core and 42 for without UNSAT core. When applying UNSAT core, the number of detected problems is not improved, but we observed that running time is slightly improved (by examination of running time for each detected problem). One reason would be that our current implementation for UNSAT core is not well organized. For identifying UNSAT cores of an UNSAT API, we evaluate all sub polynomials of the API and this process could take much time. We are planning to optimize for this step.

5.1.2 Incremental Test Data Generation

Performing a large number of test data generations affects efficiency. For instance, if we consider a polynomial constraint with 30 variables and we generate 2 test data for each variable, we have 2^{30} test data as total, which is intractable. The ideas for incremental test data generation are,

(i) an atomic polynomial inequality (API)-wise test data generation with dynamic sorting of IA-SAT APIs, and

(ii) thinning test data that does not satisfy an API.

Note that, during test data generation, test data are generated for IA-SAT APIs only because these APIs are regarded as *unknown* by IA. Let $\{f_i > 0\}$ be the set of IA-SAT APIs, and let $Var(f_i)$ be the set of variables appearing in an API f_i.

For an API-wise test data generation, an ordering of testing of IA-SAT APIs affects the efficiency. Our ideas are,

- API with a smaller variable set,

- bottleneck API w.r.t. dependency ($Var(f_i) \subseteq Var(f_j)$), and
- API with a smaller additional test data generation

have priority. To formalize them, let $DEP_{f_i} = \{f_j \mid f_j \in P \wedge Var(f_i) \subseteq Var(f_j)\}$ and $dep_{f_i} = |DEP_{f_i}|$. Then, during an API-wise test data generation, $\{f_j\}$ is dynamically sorted at the choice of next API to hold,

(a) $Var(f_i) \subset Var(f_j)$ implies $i \leq j$,

(b) dep_{f_1} is the largest, and

(c) if, for some $j < m$, $Var(f_m) \subseteq \bigcup_{i=1}^{j} Var(f_i)$ and $\forall n.\ Var(f_n) \nsubseteq \bigcup_{i=1}^{j} Var(f_i)$, then $m \leq n$,

Note that $|DEP_{f_i}|$ is denoted for cardinality of the set DEP_{f_i} and APIs are dynamically sorted as $f_1, f_2, ..., f_k$.

An API-wise test data generation requires storing previous test results of tested APIs. To reduce stored test results, test data refuting APIs are removed. When they become empty, it returns Test-UNSAT, and shifts the API that refutes all test data to the refinement.

Example 5.1.4. Let the set of IA-SAT APIs for testing be $f_1 = 2x - y^2 - 2 > 0$, $f_2 = x^2 - 1 > 0$, $f_3 = xy - yz - zx > 0$, $f_4 = u^2 - x^2y > 0$, and $f_5 = 2yv^2 - ux^2 - 1 > 0$ with $x, y, z, u, v \in (0, 2)$, and 2-random ticks be applied for testing. We have, $dep_{f_1} = 3$, $dep_{f_2} = 5$, $dep_{f_3} = 2$, $dep_{f_4} = 2$ and $dep_{f_5} = 1$. Dynamically sorted order of these APIs is f_2, f_1, f_3, f_4, f_5, shown in Figure 5.1. Testing bases on this order for generating test data, which is,

- first, $f_2 = x^2 - 1 > 0$ is chosen, since dep_{f_2} is the largest. Assume that generated test data are $\{x = 1.2, x = 0.5\}$. The satisfiable test set for $x^2 - 1 > 0$ is $\{x = 1.2\}$.
- Next $f_1 = 2x - y^2 - 2 > 0$ is chosen, since $\{y\}$ is a smaller set of additional variables and $Var(f_1) \subset Var(f_3), Var(f_4), Var(f_5)$. Assume that generated test data are $\{y = 1.4, y = 0.5\}$. The satisfiable test set $2x - y^2 - 2 > 0$ becomes $\{x = 1.2, y = 0.5\}$.

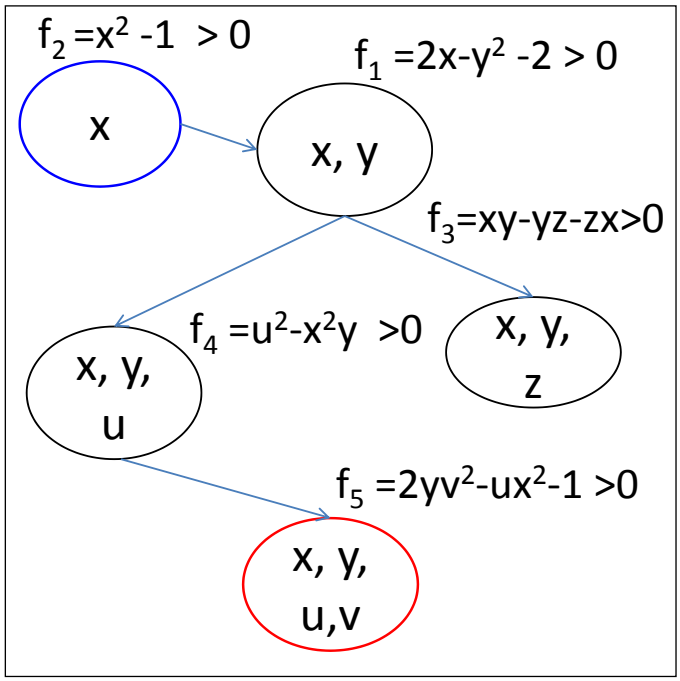

Figure 5.1: Incremental test data generation

- The API $f_3 = xy - yz - zx > 0$ is chosen (we can choose either $f_3 = xy - yz - zx > 0$ or $f_4 = u^2 - x^2y > 0$ because $Var(f_3) \subset Var(f_5)$ and $Var(f_4) \subset Var(f_5)$). Assume that generated test data are $\{z = 0.8, z = 0.3\}$ and the satisfiable test set is $\{x = 1.2, y = 0.5, z = 0.3\}$.

- For $f_4 = u^2 - x^2y > 0$, assume that test data are $\{u = 1.05, u = 0.25\}$ and the satisfiable test set is $\{x = 1.2, y = 0.5, z = 0.3, u = 1.05\}$.

- Finally, for $f_5 = 2yv^2 - ux^2 - 1 > 0$, assume that generated test data are $\{v = 0.7, v = 1.3\}$. Neither satisfies it and Test-UNSAT is reported, and $f_5 = 2yv^2 - ux^2 - 1 > 0$ is shifted to *interval decomposition*. If generated test data are $\{v = 1.13, v = 1.77\}$, $\{x = 1.2, y = 0.5, z = 0.3, u = 1.05, v = 1.77\}$ satisfies the IA-SAT APIs and testing reports SAT.

Example 5.1.4 shows that instead of generating 2^5 test data at beginning, number of test data kept in memory decreases much by incremental test data generation based on an ordering of IA-SAT APIs for testing.

Selecting Intervals to decompose	SAT		UNSAT	
	No. Problems	Time(s)	No. Problems	Time(s)
Random choice of an API	20	123.623	9	0.512
The first Test-UNSAT API	**38**	**1480.812**	**9**	**0.512**

Table 5.2: Experimental results for selecting of intervals to decompose

5.2 Strategies for Refinement

Similar to explosion of test data generation, interval decomposition may cause exponential explosion of boxes. We need to consider the choice of intervals to decompose, and how to decompose an interval.

5.2.1 Selecting Intervals to Decompose

The choice of intervals to decompose consists of two steps.

(a) Choose an API such that its variables are candidates for interval decomposition.

(b) Among variables of the selected API, choose influential ones.

(a) follows incremental test data generation in Section 5.1.2 where APIs are sorted based on their dependencies on variables. When an API $f_j > 0$ firstly refutes all generated test data, testing reports Test-UNSAT. Such an API $f_j > 0$ is called *the first Test-UNSAT API*. Then, variables appearing in f_j are candidates for interval decompositions, since f_j is a direct cause of Test-UNSAT. In Example 5.1.4, the Test-UNSAT API $2yv^2 - ux^2 - 1 > 0$ is reported with $\{v = 0.7, v = 1.3\}$, and $x, y, u, v \in (0, 2)$ (variables of the first Test-UNSAT API) become candidates for interval decomposition.

For (b), among variables in the selected API $f_j > 0$, we further filter variables that have sensitivity (Example 4.1.20) beyond a threshold, since they are expected to be more influential. Sensitivity is detected by previous IA-SAT detection phase.

Among presented strategies, only (a) is implemented in our current solver **raSAT** ((b) is not yet implemented), and we select all variables in the first Test-UNSAT API for interval decomposition.

Table 5.2 shows experimental results when we apply two different selections for selecting an API whose variables are candidate for interval decomposition in **raSAT**. One is

random choice of an API among IA-SAT APIs and the other selects the first Test-UNSAT API. Then, all variables in the selected API are applied for interval decomposition. The problems chosen for the experiment are 151 inequality problems of *Zankl* family (in the division QF_NRA of SMT-LIB [2]). The number of solved problems and their total running time are indicated in SAT and UNSAT columns of Table 5.2.

Selecting the first Test-UNSAT API gives a better result than random choice of an API, i.e., 38 SAT problems are detected by selecting the first Test-UNSAT API, only 20 SAT problems are detected by random choice. The experiment shows that choice of variables for interval decomposition affects efficiency, and in **raSAT**, we apply a guidance from testing results (the first Test-UNSAT API), which would be useful because,

- an API that is more difficult to find SAT instances, i.e., a Test-UNSAT API, has high priority for examination. If the API is UNSAT, **raSAT** only applies interval decomposition for its variables without considering others. If it is SAT, **raSAT** quickly narrows its SAT domain.

- UNSAT domain is quickly removed because interval decomposition for variables of Test-UNSAT APIs helps to identify UNSAT domain.

Another SMT solver iSAT [12] for polynomial constraints, applies both interval arithmetic (classical interval) and interval decomposition too, unfortunately it is not clear in iSAT how variables are chosen for interval decomposition. Choice of variables for interval decomposition is quite important when solving problems with a large number of variables, i.e., 30, 40, > 100 variables, such as problems of Zankl family.

5.2.2 Interval Decomposition

We present below 3 strategies for decomposing an interval into smaller intervals, which are *balanced*, *monotonic*, and *tick* decomposition.

Balanced Decomposition

Balanced decomposition decomposes an interval into two intervals exactly half.

Definition 5.2.1. For an interval $x \in (a, b)$, a balanced decomposition is

$$D_b(x \in (a, b)) = \{x \in (a, \frac{a+b}{2}), x \in (\frac{a+b}{2}, b)\}$$

For example, $x \in (0,2)$ is decomposed into $x \in (0,1)$ and $x \in (1,2)$ by balanced decomposition, and we add the following clauses to the SAT solver for encoding interval decomposition,

$$\begin{aligned} & x \in (0,2) \Rightarrow x \in (0,1) \vee x \in (1,2) \\ \wedge \quad & x \in (0,1) \Rightarrow x \in (0,2) \\ \wedge \quad & x \in (1,2) \Rightarrow x \in (0,2) \\ \wedge \quad & (\neg(x \in (0,1)) \vee \neg(x \in (1,2))) \end{aligned}$$

Note that $\Rightarrow$ is denoted for *implication* and $u \Rightarrow v$ is equivalent with $\neg u \vee v$.

Monotonic Decomposition

Monotonic decomposition introduces a bias δ to an interval decomposition, if a value of a corresponding variable monotonically affects on a value of a polynomial.

Definition 5.2.2. Let $f(x_1, \cdots, x_k)$ be a polynomial, a variable x_i $(1 \leq i \leq k)$ is

- *monotonically increasing* in f if

$$\forall x_i' \geq x_i'' \text{ implies } f(x_1, \cdots, x_i', \cdots, x_k) \geq f(x_1, \cdots, x_i'', \cdots, x_k), \text{ and}$$

- *monotonically decreasing* in f if

$$\forall x_i' \geq x_i'' \text{ implies } f(x_1, \cdots, x_i', \cdots, x_k) \leq f(x_1, \cdots, x_i'', \cdots, x_k).$$

Pos_f and Neg_f denote for the sets of monotonically increasing and decreasing variables of a polynomial f, respectively.

For a polynomial $f = 2yv^2 - ux^2 - 1$ where $x, y, u, v \in (0, \infty)$, $Pos_f = \{y, v\}$ and $Neg_f = \{x, u\}$.

Definition 5.2.3. Let $x \in (a,b)$ and $\delta < b - a$ for a bound δ. A monotonic decomposition is,

$$D_m(x \in (a,b)) = \begin{cases} \{x \in (a, b-\delta), x \in (b-\delta, b)\} & \textit{if } x \in Pos_f \\ \{x \in (a, a+\delta), x \in (a+\delta, b)\} & \textit{if } x \in Neg_f \\ \{x \in (a, \frac{a+b}{2}), x \in (\frac{a+b}{2}, b)\} & \textit{otherwise} \end{cases}$$

Example 5.2.4. In Example 5.1.4, if the API $f = 2yv^2 - ux^2 - 1 > 0$ is Test-UNSAT under $I = x \in (0,2) \wedge y \in (0,2) \wedge u \in (0,2) \wedge v \in (0,2)$, the following clauses are added to the SAT solver for monotonic decomposition (with assumption $\delta = 0.25$),

- $x \in Neg_f$ then $x \in (0,2)$ is decomposed into $x \in (0,0.25)$ and $x \in (0.25,2)$,

$$\begin{aligned} & x \in (0,2) \Rightarrow x \in (0,0.25) \vee x \in (0.25,2) \\ \wedge \quad & x \in (0,0.25) \Rightarrow x \in (0,2) \\ \wedge \quad & x \in (0.25,2) \Rightarrow x \in (0,2) \\ \wedge \quad & (\neg(x \in (0,0.25)) \vee \neg(x \in (0.25,2))) \end{aligned}$$

- $y \in Pos_f$ then $y \in (0,2)$ is decomposed into $y \in (0,1.75)$ and $y \in (1.75,2)$,

$$\begin{aligned} & y \in (0,2) \Rightarrow y \in (0,1.75) \vee y \in (1.75,2) \\ \wedge \quad & y \in (0,1.75) \Rightarrow y \in (0,2) \\ \wedge \quad & y \in (1.75,2) \Rightarrow y \in (0,2) \\ \wedge \quad & (\neg(y \in (0,1.75)) \vee \neg(y \in (1.75,2))) \end{aligned}$$

- $u \in Neg_f$ then $u \in (0,2)$ is decomposed into $u \in (0,0.25)$ and $u \in (0.25,2)$,

$$\begin{aligned} & u \in (0,2) \Rightarrow u \in (0,0.25) \vee u \in (0.25,2) \\ \wedge \quad & u \in (0,0.25) \Rightarrow u \in (0,2) \\ \wedge \quad & u \in (0.25,2) \Rightarrow u \in (0,2) \\ \wedge \quad & (\neg(u \in (0,0.25)) \vee \neg(u \in (0.25,2))) \end{aligned}$$

- $v \in Pos_f$ then $v \in (0,2)$ is decomposed into $v \in (0,1.75)$ and $v \in (1.75,2)$,

$$\begin{aligned} & v \in (0,2) \Rightarrow v \in (0,1.75) \vee v \in (1.75,2) \\ \wedge \quad & v \in (0,1.75) \Rightarrow v \in (0,2) \\ \wedge \quad & v \in (1.75,2) \Rightarrow v \in (0,2) \\ \wedge \quad & (\neg(v \in (0,1.75)) \vee \neg(v \in (1.75,2))) \end{aligned}$$

We apply δ in Definition 5.2.3 (heuristic rule) as the bias δ, by regarding δ as a unit of searching. For a balanced decomposition, the SAT solver will choose an arbitrary combination of input ranges. However, for a monotonic decomposition, we would like to force the SAT solver to choose a narrower sub-interval (i.e., $(b-\delta, b)$ for Pos_f, and $(a, a+\delta)$ for Neg_f), which makes upper and lower bounds of the polynomial f increase, and provides more chances to lead f satisfiable. In Example 5.2.4, the choice is $x \in \mathbf{(0,0.25)} \wedge y \in \mathbf{(1.75,2)} \wedge u \in \mathbf{(0,0.25)} \wedge v \in \mathbf{(1.75,2)}$ for the next evaluation by IA and testing. MiniSat 2.2 chooses literals by the "activity" measure, and we manually increase the activity of literals corresponding to a narrowed sub-interval.

Tick Decomposition

Tick decomposition divides an interval into two or three sub intervals based on a given point (tick) inside the interval.

Definition 5.2.5. Let $x \in (a, b)$, $\delta < b - a$ for a bound δ and a tick $t \in (a, b)$, d_p, d_n and d_{pn} are defined as follows:

$$d_p(x \in (a,b)) = \begin{cases} \{x \in (a,t) \vee x \in (t, t+\delta) \vee x \in (t+\delta, b)\} & \textit{if } t+\delta < b \\ \{x \in (a,t) \vee x \in (t,b)\} & \textit{otherwise} \end{cases}$$

$$d_n(x \in (a,b)) = \begin{cases} \{x \in (a,t-\delta) \vee x \in (t-\delta, t) \vee x \in (t,b)\} & \textit{if } t-\delta > a \\ \{x \in (a,t) \vee x \in (t,b)\} & \textit{otherwise} \end{cases}$$

$$d_{pn}(x \in (a,b)) = \begin{cases} \{x \in (a,t+0.5\delta) \vee x \in (t+0.5\delta, b)\} & \textit{if } t-0.5\delta \leq a \\ \{x \in (a,t-0.5\delta) \vee x \in (t-0.5\delta, b)\} & \textit{if } t+0.5\delta \geq b \\ \{x \in (a,t-0.5\delta) \vee x \in (t-0.5\delta, t+0.5\delta) \vee x \in (t+0.5\delta, b)\} & \textit{otherwise} \end{cases}$$

Definition 5.2.6. Let $x \in (a, b)$, $\delta < b - a$ for a bound δ and a tick $t \in (a, b)$. Tick decomposition is,

$$D_t(x \in (a,b)) = \begin{cases} d_p & \textit{if } x \in Pos_f \\ d_n & \textit{if } x \in Neg_f \\ d_{pn} & \textit{otherwise} \end{cases}$$

Example 5.2.7. In Example 5.1.4, assume that the API $f = 2yv^2 - ux^2 - 1 > 0$ is Test-UNSAT under $I = x \in (0,2) \wedge y \in (0,2) \wedge u \in (0,2) \wedge v \in (0,2)$, $\delta = 0.25$ and ticks are given by $t_x = 1.2, t_y = 0.5, t_u = 1.05$ and $t_v = 1.3$ (a test data in Example 5.1.4). Because $x, u \in Neg_f$ and $y, v \in Pos_f$, clauses for tick decomposition are,

- $x \in (0,2)$ is decomposed into $x \in (0, 0.95)$, $x \in (0.95, 1.2)$ and $x \in (1.2, 2)$,

$$
\begin{aligned}
& x \in (0,2) \Rightarrow x \in (0,0.95) \vee x \in (0.95,1.2) \vee x \in (1.2,2) \\
\wedge \quad & x \in (0,0.95) \Rightarrow x \in (0,2) \\
\wedge \quad & x \in (0.95,1.2) \Rightarrow x \in (0,2) \\
\wedge \quad & x \in (1.2,2) \Rightarrow x \in (0,2) \\
\wedge \quad & (\neg(x \in (0,0.95)) \vee \neg(x \in (0.95,1.2))) \\
\wedge \quad & (\neg(x \in (0,0.95)) \vee \neg(x \in (1.2,2))) \\
\wedge \quad & (\neg(x \in (0.95,1.2)) \vee \neg(x \in (1.2,2)))
\end{aligned}
$$

- $y \in (0,2)$ is decomposed into $y \in (0, 0.5)$, $y \in (0.5, 0.75)$ and $y \in (0.75, 2)$,

$$
\begin{aligned}
& y \in (0,2) \Rightarrow y \in (0,0.5) \vee y \in (0.5,0.75) \vee y \in (0.75,2) \\
\wedge \quad & y \in (0,0.5) \Rightarrow y \in (0,2) \\
\wedge \quad & y \in (0.5,0.75) \Rightarrow y \in (0,2) \\
\wedge \quad & y \in (0.75,2) \Rightarrow y \in (0,2) \\
\wedge \quad & (\neg(y \in (0,0.5)) \vee \neg(y \in (0.5,0.75))) \\
\wedge \quad & (\neg(y \in (0,0.5)) \vee \neg(y \in (0.75,2))) \\
\wedge \quad & (\neg(y \in (0.5,0.75)) \vee \neg(y \in (0.75,2)))
\end{aligned}
$$

- $u \in (0, 2)$ is decomposed into $u \in (0, 0.8)$, $u \in (0.8, 1.05)$ and $u \in (1.05, 2)$,

$$
\begin{aligned}
& u \in (0,2) \Rightarrow u \in (0,0.8) \vee u \in (0.8,1.05) \vee u \in (1.05,2) \\
\wedge \quad & u \in (0,0.8) \Rightarrow u \in (0,2) \\
\wedge \quad & u \in (0.8,1.05) \Rightarrow u \in (0,2) \\
\wedge \quad & u \in (1.05,2) \Rightarrow u \in (0,2) \\
\wedge \quad & (\neg(u \in (0,0.8)) \vee \neg(u \in (0.8,1.05))) \\
\wedge \quad & (\neg(u \in (0,0.8)) \vee \neg(u \in (1.05,2))) \\
\wedge \quad & (\neg(u \in (0.8,1.05)) \vee \neg(u \in (1.05,2)))
\end{aligned}
$$

- $v \in (0, 2)$ is decomposed into $v \in (0, 1.3)$, $v \in (1.3, 1.55)$ and $v \in (1.55, 2)$,

$$
\begin{aligned}
& v \in (0,2) \Rightarrow v \in (0,1.3) \vee v \in (1.3,1.55) \vee v \in (1.55,2) \\
\wedge \quad & v \in (0,1.3) \Rightarrow v \in (0,2) \\
\wedge \quad & v \in (1.3,1.55) \Rightarrow v \in (0,2) \\
\wedge \quad & v \in (1.55,2) \Rightarrow v \in (0,2) \\
\wedge \quad & (\neg(v \in (0,1.3)) \vee \neg(v \in (1.3,1.55))) \\
\wedge \quad & (\neg(v \in (0,1.3)) \vee \neg(v \in (1.55,2))) \\
\wedge \quad & (\neg(v \in (1.3,1.55)) \vee \neg(v \in (1.55,2)))
\end{aligned}
$$

In implementation of **raSAT**, a given tick is the test data that makes the first Test-UNSAT API f_j be max value. Intuitionally, we would like to force the SAT solver to choose a narrower sub-interval around the max value test data (i.e., $(t, t + \delta)$ or (t, b) for Pos_f, $(t - \delta, t)$ or (a, t) for Neg_f). In Example 5.2.7, the choice is $x \in$ **(0.95,1.2)** $\wedge\, y \in$ **(0.5,0.75)** $\wedge\, u \in$ **(0.8,1.05)** $\wedge\, v \in$ **(1.3,1.55)** for the next evaluation by IA and testing.

We also compare results when applying different strategies for interval decomposition. Table 5.3 shows experimental results for 151 problems of Zankl family. The first column indicates 3 strategies, which are *Balanced*, *Monotonic* and *Tick Decomposition*. The number of solved problems (SAT/UNSAT) and their total running time are shown in the other columns. Though *Monotonic Decomposition* gives the best results, the difference

Interval Decomposition	**SAT**		**UNSAT**	
	No. Problems	Time(s)	No. Problems	Time(s)
Balanced	35	1490.759	9	1.362
Monotonic	**38**	**1480.812**	**9**	**0.512**
Tick	24	431.475	9	0.512

Table 5.3: Experimental results for different strategies of interval decomposition

with *Balanced Decomposition* is not large. We also plan to investigate other choices, i.e., adjusting length of smaller intervals, identifying UNSAT domain, etc.

Chapter 6

The SMT Solver raSAT and Experiments

6.1 Design Framework of raSAT

We implement the SMT solver **raSAT** following to the DPLL(T) procedure in Chapter 3 by instantiating *Interval Arithmetic* (IA) and *Testing* as $O.T$ and $U.T$, respectively. Figure 6.1 shows its framework.

Initial input constraint is represented in the form of $F = I \wedge P$ where I is an interval constraint, $I = (x_1 \in (a_1, b_1)) \wedge \cdots \wedge (x_n \in (a_n, b_n))$, presented for a target domain of searching, and polynomial constraint P is in the form of $\bigwedge_{i=1}^{m} f_i(x_1, \cdots, x_n) > 0$. Note that P is conjunction of f_i, then only I is sent to SAT solver. After processes of interval decomposition or removing UNSAT domain (by learning clauses in SAT solver), I becomes a CNF formula represented for current searching domain.

We use **MiniSat2.2** as backend SAT solver. It is asked to choose a box represented for a combination of input ranges for all variables, which is first evaluated by IA. If IA informs IA-UNSAT, the combination is sent to **UNSAT cores** in *Theory Propagation* for computing minimal combinations that make IA-UNSAT. The very lazy theory learning is applied for learning clauses of minimal combinations obtained from UNSAT cores, which is aimed at removing UNSAT domain from the target domain. Then the SAT solver chooses another box. If IA informs IA-VALID, **raSAT** terminates and outputs SAT. Otherwise, testing is applied.

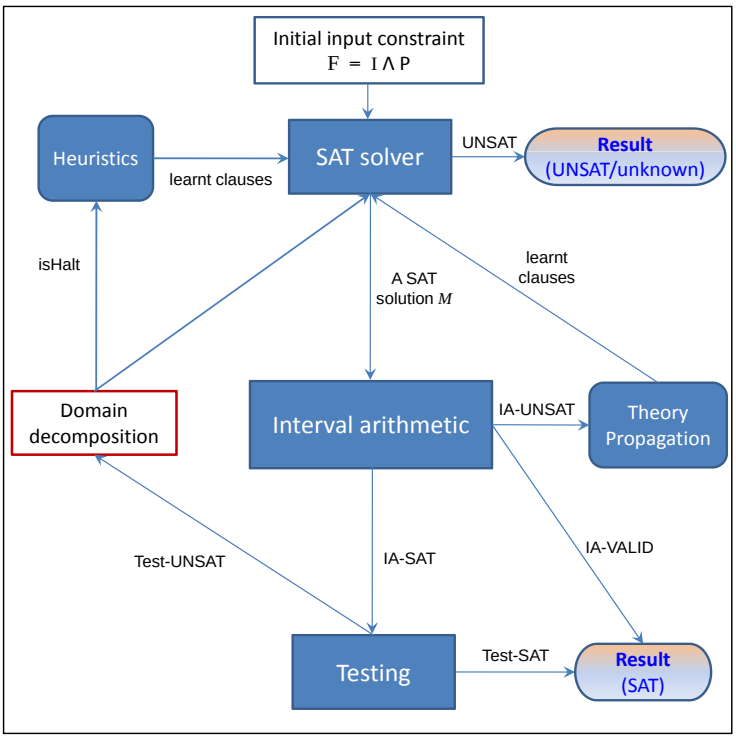

Figure 6.1: Framework of **raSAT**

Testing generates test data from the box (a given combination of input ranges for all variables). Testing will stop when it results Test-SAT, i.e., a SAT solution is found. Otherwise it finds the first Test-UNSAT API by *incremental test data generation* in Section 5.1.2. It is sent to *domain decomposition* for refinements on the box, i.e., *interval decomposition* is applied to decomposing the box into smaller boxes.

If the box becomes small enough, i.e., less than a given threshold, *Heuristic* is applied by using *heuristic rule* to remove that box. Once heuristic rules are used, **raSAT** concludes *unknown* when the SAT solver informs UNSAT. If they are never used, **raSAT** concludes UNSAT when the same occurs.

Example 6.1.1. Figure 6.2 describes process of solving the polynomial inequality constraint $F = (x \in (-1,3) \wedge y \in (-1,3)) \wedge (x^3 - x^2 + y - 1.99 > 0)$. **raSAT** initially searches on the box $x \in (-1,3) \wedge y \in (-1,3)$, and executes a refinement loop for searching on their sub boxes.

We present process of **raSAT** for *balanced* and *monotonic* decomposition by the DPLL(T) rules for approximation refinement. P is denoted for the polynomial constraint $x^3 - x^2 + y - 1.99 > 0$.

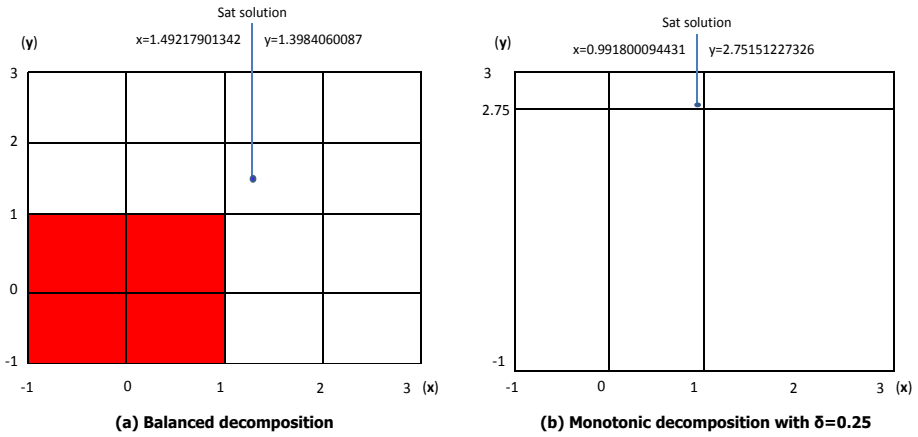

Figure 6.2: Interval decompositions by **raSAT** for Example 6.1.1

Note that $\Longrightarrow_{VL}$ (SAT solver) is the rule that forces the SAT solver to choose a full assignment (a combination of input ranges). For an interval decomposition, i.e., $x \in (-1, 3)$ is decomposed into $x \in (-1, 1) \vee x \in (1, 3)$, in implementation we add four clauses below to the SAT solver,

$$
\begin{aligned}
& x \in (-1,3) \Rightarrow x \in (-1,1) \vee x \in (1,3) \\
\wedge \quad & x \in (-1,1) \Rightarrow x \in (-1,3) \\
\wedge \quad & x \in (1,3) \Rightarrow x \in (-1,3) \\
\wedge \quad & (\neg(x \in (-1,1)) \vee \neg(x \in (1,3)))
\end{aligned}
$$

But for simplicity in presentation of the DPLL(T) rules, we only add the clause $x \in (-1, 1) \vee x \in (1, 3)$ for the 4 clauses.

Balanced decomposition: Disjoin boxes applied balanced decomposition are shown in the left of Figure 6.2. The red boxes are detected as UNSAT (IA-UNSAT).

$$
\begin{array}{rll}
\emptyset & \| \ (x \in (-1,3)) \wedge (y \in (-1,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_0 & \| \ (x \in (-1,3)) \wedge (y \in (-1,3)) \wedge P & \Longrightarrow_{VL} \text{(refinement)} \\
\emptyset & \| \cdots \wedge (x \in (-1,1) \vee x \in (1,3)) \wedge (y \in (-1,1) \vee y \in (1,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_1 & \| \cdots \wedge (x \in (-1,1) \vee x \in (1,3)) \wedge (y \in (-1,1) \vee y \in (1,3)) \wedge P & \Longrightarrow_{VL} \text{(refinement)} \\
\emptyset & \| \cdots \wedge (x \in (-1,0) \vee x \in (0,1)) \wedge (y \in (-1,0) \vee y \in (0,1)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_2 & \| \cdots \wedge (x \in (-1,0) \vee x \in (0,1)) \wedge (y \in (-1,0) \vee y \in (0,1)) \wedge P & \Longrightarrow_{VL} \text{(Learning)} \\
\emptyset & \| \cdots \wedge (\neg(x \in (-1,0)) \vee \neg(y \in (-1,0))) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_3 & \| \cdots \wedge (\neg(x \in (-1,0)) \vee \neg(y \in (-1,0))) \wedge P & \Longrightarrow_{VL} \text{(Learning)} \\
\emptyset & \| \cdots \wedge (\neg(x \in (-1,0)) \vee \neg(y \in (0,1))) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_4 & \| \cdots \wedge (\neg(x \in (-1,0)) \vee \neg(y \in (0,1))) \wedge P & \Longrightarrow_{VL} \text{(Learning)} \\
\emptyset & \| \cdots \wedge (\neg(x \in (0,1)) \vee \neg(y \in (-1,0))) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_5 & \| \cdots \wedge (\neg(x \in (0,1)) \vee \neg(y \in (-1,0))) \wedge P & \Longrightarrow_{VL} \text{(Learning)} \\
\emptyset & \| \cdots \wedge (\neg(x \in (0,1)) \vee \neg(y \in (0,1))) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_6 & \| \cdots \wedge (\neg(x \in (0,1)) \vee \neg(y \in (0,1))) \wedge P & \Longrightarrow_{VL} \text{(refinement)} \\
\emptyset & \| \cdots \wedge (x \in (1,2) \vee x \in (2,3)) \wedge (y \in (1,2) \vee y \in (2,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_7 & \| \cdots \wedge (x \in (1,2) \vee x \in (2,3)) \wedge (y \in (1,2) \vee y \in (2,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT rule)} \\
 & SAT & \text{Model found}
\end{array}
$$

where M_i chosen by the SAT solver are,

- $M_0 = (x \in (-1,3)) \wedge (y \in (-1,3))$
- $M_1 = (x \in (-1,1)) \wedge (y \in (-1,1))$
- $M_2 = (x \in (-1,0)) \wedge (y \in (-1,0))$
- $M_3 = (x \in (-1,0)) \wedge (y \in (0,1))$
- $M_4 = (x \in (0,1)) \wedge (y \in (-1,0))$
- $M_5 = (x \in (0,1)) \wedge (y \in (0,1))$
- $M_6 = (x \in (1,3)) \wedge (y \in (1,3))$

- $M_7 = (x \in (1,2)) \wedge (y \in (1,2))$

When the SAT solver chooses $M_7 = (x \in (1,2)) \wedge (y \in (1,2))$, IA results IA-SAT, and testing finally finds a satisfiable test data $x = 1.49217901342$ and $y = 1.3984060087$ (Test-SAT). Then, **raSAT** returns SAT.

Monotonic decomposition: Similar process but with monotonic decomposition is described in the right of Figure 6.2 (with assumption $\delta = 0.25$).

$$
\begin{array}{rll}
\emptyset & \| \ (x \in (-1,3)) \wedge (y \in (-1,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_0 & \| \ (x \in (-1,3)) \wedge (y \in (-1,3)) \wedge P & \Longrightarrow_{VL} \text{(refinement)} \\
\emptyset & \| \ \cdots \wedge (x \in (-1,1) \vee x \in (1,3)) \wedge (y \in (-1,2.75) \vee y \in (2.75,3)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_1 & \| \ \cdots \wedge (x \in (-1,1) \vee x \in (1,3)) \wedge (y \in (-1,2.75) \vee y \in (2.75,3)) \wedge P & \Longrightarrow_{VL} \text{(refinement)} \\
\emptyset & \| \ \cdots \wedge (y \in (-1,2.75) \vee y \in (2.75,3)) \wedge (x \in (-1,0) \vee x \in (0,1)) \wedge P & \Longrightarrow_{VL} \text{(SAT solver)} \\
M_2 & \| \ \cdots \wedge (y \in (-1,2.75) \vee y \in (2.75,3)) \wedge (x \in (-1,0) \vee x \in (0,1)) \wedge P & \Longrightarrow_{VL} \text{(SAT rule)} \\
 & SAT & \text{Model found}
\end{array}
$$

where M_i chosen by the SAT solver are,

- $M_0 = (x \in (-1,3)) \wedge (y \in (-1,3))$
- $M_1 = (x \in (-1,1)) \wedge (y \in (2.75,3))$
- $M_2 = (x \in (0,1)) \wedge (y \in (2.75,3))$

The SAT solver chooses $M_2 = x \in (0,1) \wedge y \in (2.75,3)$ and testing finds a satisfiable test data with $x = 0.991800094431$ and $y = 2.75151227326$ (Test-SAT). With monotonic decomposition, **raSAT** finds a satisfiable instance with fewer decompositions.

6.2 Experiments

In experiments, we first apply different measures, i.e., high degrees, number of variables, number of APIs (atomic polynomial inequalities) to compare our solver **raSAT** with other SMT solvers, and then we test on problems of the division QF_NRA (Quantifier Free of Nonlinear Real Arithmetic) in the benchmarks of SMT-LIB [2].

We compare **raSAT** with **Z3 4.3**. Z3 (version 3.1) is the winner of SMT competition 2011 for QF_NRA and the latest version (**Z3 4.3**) is also called by another name **nlSAT** [14], which is believed as the strongest SMT solver for non-linear arithmetic.

In fact, it is possible to apply classical interval (CI) for IA in **raSAT**, but we apply affine intervals (i.e., AF_2, CAI) coming from following reasons.

- Affine intervals could *improve precision* for estimating bounds of polynomials. They are not always better than CI, however they are more likely for computing of dependencies.
- Affine intervals are aimed at *guiding refinements* and *testing* from *sensitivity* of variables, i.e., high degrees, large coefficients. However it is not yet implemented in current **raSAT**, but it is planned for next implementation.

We apply *2-random ticks* for testing, *Test-UNSAT* of testing and *monotonic decomposition* for refinements are used. We do not apply UNSAT core computation in these experiments. All tests are run on a system with Intel Core Duo L7500 1.6 GHz and 2 GB of RAM.

6.2.1 Experiments on Different Measures

The first and the second measures apply for the problems of the form

$$\psi = \sum_{i=1}^{k} x_i^n < 1 \wedge \sum_{i=1}^{k} (x_i - r)^n < 1 \tag{6.1}$$

For experiments on the problems 6.1, we adjust values of r based on the threshold $\sqrt[n]{\frac{1}{k}}$, which is the threshold separating SAT and UNSAT problems when k and n are fixed. If r is close to the threshold, the problems are more difficult to decide and vice versa. Figure 6.3 and 6.4 demonstrate the choices of r. In our experiments we choose values of r for difficult SAT/UNSAT problems such that $|r - \sqrt[n]{\frac{1}{k}}| < 0.01$.

For settings of **raSAT**, the *unit searching* δ is set to 0.005 and initial interval constraint is presented as $\bigwedge_i x_i \in (-1, 1)$.

The first measure: degree of polynomials

SAT/UNSAT	k	n	r	Time(s)	
				Z3 4.3	raSAT
SAT	2	4	1.68	0.080	0.062
UNSAT	2	4	1.69	0.050	0.328
SAT	2	6	1.78	0.390	0.250
UNSAT	2	6	1.79	0.300	0.375
SAT	2	8	1.83	1.330	0.265
UNSAT	2	8	1.84	0.580	0.328
SAT	2	10	1.86	4.530	0.140
UNSAT	2	10	1.87	125.000	0.796
SAT	2	12	1.88	0.360	0.140
UNSAT	2	12	1.89	40.280	1.390
SAT	2	14	1.90	0.480	0.296
UNSAT	2	14	1.91	78.730	0.531
SAT	2	16	1.91	2.250	0.109
UNSAT	2	16	1.92	174.000	0.484
SAT	2	18	1.92	289.110	0.562
UNSAT	2	18	1.93	391.670	0.765
SAT	2	20	1.93	1259.560	1.468
UNSAT	2	20	1.94	1650.860	0.921
SAT	2	22	1.93	$? > 3600$	0.437
UNSAT	2	22	1.94	$? > 3600$	3.203

Table 6.1: Experimental results for $\psi = x_1^n + x_2^n < 1 \wedge (x_1 - r)^n + (x_2 - r)^n < 1$

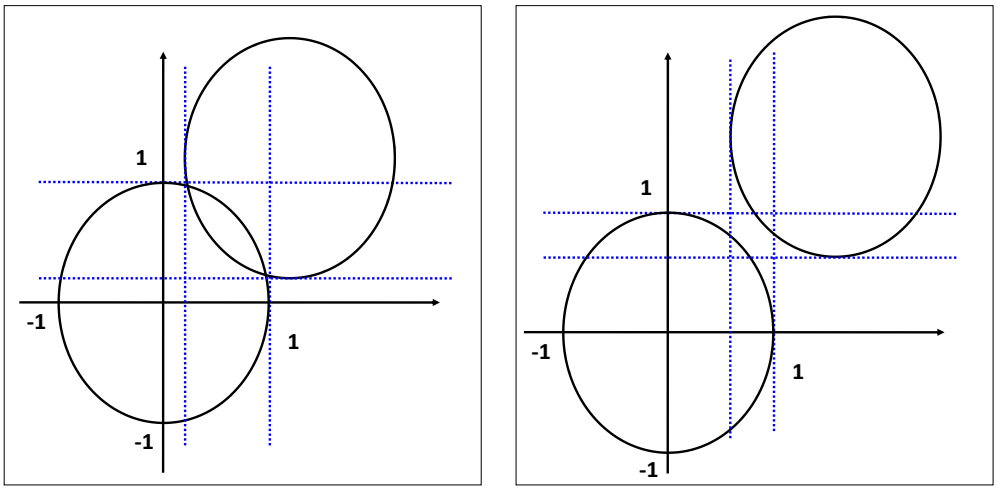

Figure 6.3: The choices of r are far from the threshold $\sqrt[n]{\frac{1}{k}}$

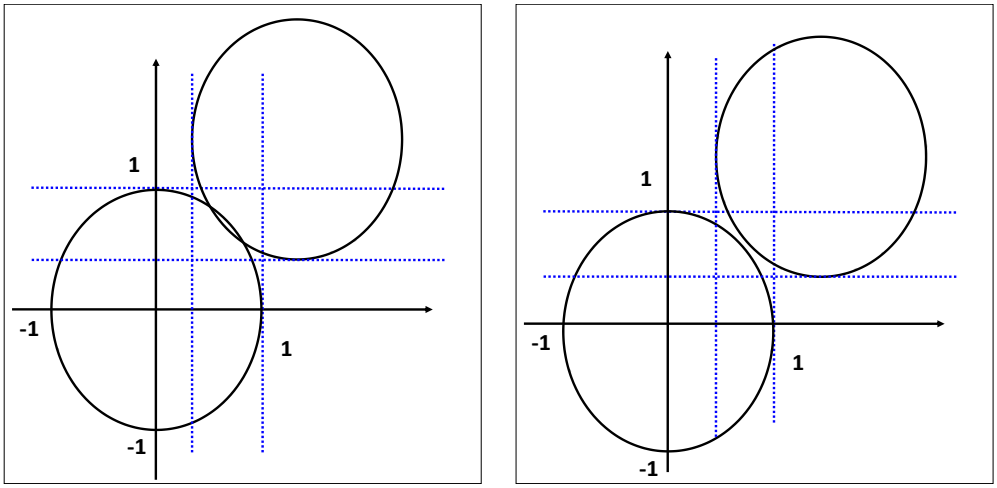

Figure 6.4: The choices of r are close to the threshold $\sqrt[n]{\frac{1}{k}}$

We first show comparison of **raSAT** and **Z3 4.3** in Table 6.1 for the problems,

$$\psi = x_1^n + x_2^n < 1 \wedge (x_1 - r)^n + (x_2 - r)^n < 1$$

when we fix the number of variables to 2 ($k = 2$), increase degrees n (i.e., $n = 4, 6, 8, ..., 22$), and modify values of r.

In Table 6.1, the first column indicates problems that are either SAT or UNSAT, the next are columns of k, n, and r, respectively. We compare running time in seconds of **Z3 4.3** and **raSAT** for each problem in the last two columns. For the degree 22, the results of **Z3 4.3** are $''? > 3600''$, which means that **Z3 4.3** didn't respond in 3600 seconds. Note

that we didn't set timeout in the first experiment.

For the first experiment, we aim at seeing how **raSAT** and **Z3 4.3** are performed for problems with fewer variables (only 2) but high degrees. **raSAT** shows interesting results beyond **Z3 4.3**, which are,

- **raSAT** reports results very fast, i.e., its solving time are almost around 1 second and the longest running time is 3.203 seconds for the degree 22,
- **raSAT** does well for high degrees of 2 variables, while **Z3 4.3** suffers from degree increasing, i.e., 289.110 second for the degree 18 (SAT), 1259.560 seconds for the degree 20 (SAT), and no response in 1 hour for the degree 22 (SAT and UNSAT).
- For each degree, we shift values of r a little from SAT to UNSAT (i.e., r is shifted from 1.86 to 1.87 for the degree 10, from 1.92 to 1.93 for the degree 18), **raSAT** can decide for both SAT and UNSAT problems in *short time*.

The second measure: number of variables

The problems in the second measure are taken from the formula (6.1) when we increase their number of variables (e.g., from 3 to 6) and their degrees (e.g., 4, 6, 8). We aim at seeing how **raSAT** and **Z3 4.3** work when both dimensions and degrees are enlarged. Results are shown in Table 6.2, which are,

- **raSAT** still outperforms **Z3 4.3**, i.e., 12 problems are solved by **raSAT**, and only 2 problems are solved by **Z3 4.3** in total 20 problems,
- when dimension increasing, **raSAT** can detect for SAT problems but their running time are increased much, i.e., from 0.390 seconds for 4 variables to 51.578 seconds for 5 variables of the degree 4,
- however, **raSAT** suffers from increasing number of variables (dimensions), especially for UNSAT problems because of explosion on number of boxes. For UNSAT problems, **raSAT** could extract all small boxes recognized by the unit searching $\delta = 0.005$, and proves them UNSAT by interval arithmetic.

In the second experiment, timeout is set to 600 seconds for each problem.

SAT/UNSAT	k	n	r	Time(s)	
				Z3 4.3	raSAT
SAT	3	4	1.51	0.030	0.027
UNSAT	3	4	1.52	10.560	*timeout*
SAT	4	4	1.41	*timeout*	0.390
UNSAT	4	4	1.42	*timeout*	*timeout*
SAT	5	4	1.33	*timeout*	51.578
UNSAT	5	4	1.34	*timeout*	*timeout*
SAT	6	4	1.27	*timeout*	111.031
UNSAT	6	4	1.28	*timeout*	*timeout*
SAT	3	6	1.66	*timeout*	0.890
UNSAT	3	6	1.67	*timeout*	62.765
SAT	4	6	1.58	*timeout*	1.156
UNSAT	4	6	1.59	*timeout*	*timeout*
SAT	5	6	1.52	*timeout*	73.937
UNSAT	5	6	1.53	*timeout*	*timeout*
SAT	6	6	1.48	*timeout*	239.968
UNSAT	6	6	1.49	*timeout*	*timeout*
SAT	3	8	1.74	*timeout*	3.125
UNSAT	3	8	1.75	*timeout*	37.156
SAT	4	8	1.68	*timeout*	69.843
UNSAT	4	8	1.69	*timeout*	*timeout*

Table 6.2: Experimental results for $\psi = \sum_{i=1}^{k} x_i^n < 1 \wedge \sum_{i=1}^{k} (x_i - r)^n < 1$

SAT/UNSAT	k	n	r	Time(s)	
				Z3 4.3	raSAT
SAT	3	6	1.78	*timeout*	0.171
UNSAT	3	6	1.79	0.280	0.796
SAT	5	6	1.78	*timeout*	0.375
UNSAT	5	6	1.79	0.280	0.640
SAT	7	6	1.78	*timeout*	0.765
UNSAT	7	6	1.79	0.250	0.734
SAT	9	6	1.78	*timeout*	2.671
UNSAT	9	6	1.79	0.300	1.921
SAT	11	6	1.78	*timeout*	3.328
UNSAT	11	6	1.79	0.220	1.343
SAT	13	6	1.78	*timeout*	4.460
UNSAT	13	6	1.79	0.300	1.875
SAT	15	6	1.78	*timeout*	6.640
UNSAT	15	6	1.79	0.300	2.265

Table 6.3: Experimental results for $\psi = \psi_1 \wedge \psi_2$

The third measure: number of APIs

The third measure is on increasing number of APIs (atomic polynomial inequalities) and its problems are taken from the formula $\psi = \psi_1 \wedge \psi_2$ where,

- $\psi_1 = x_0^n + x_1^n < 1 \wedge x_1^n + x_2^n < 1 \wedge \cdots \wedge x_k^n + x_0^n < 1$
- $\psi_2 = (x_0 - r)^n + (x_1 - r)^n < 1 \wedge (x_1 - r)^n + (x_2 - r)^n < 1 \wedge \cdots \wedge (x_k - r)^n + (x_0 - r)^n < 1$

The initial interval constraints are set as $\bigwedge_i x_i \in (-1, 1)$ and timeout is set in 600 seconds. We fix the degree to 6 ($n = 6$) and adjust r such that $|r - \sqrt[n]{\frac{1}{2}}| < 0.01$.

Comparison results of **Z3 4.3** and **raSAT** are shown in Table 6.3. The number of APIs is increased from 3 to 15, which is shown in the column k. The last 2 columns show running time of **Z3 4.3** and **raSAT**. While **Z3 4.3** reports timeout for 7 problems (among 14 as total and 600 seconds for timeout), **raSAT** can detect all of them. The longest running time is 6.640 seconds for the problems of 15 APIs. By this measure on number of APIs, **raSAT** still outperforms **Z3 4.3**.

Observation

From the experiments on three measures, we have some observations that are,

- **Z3 4.3** meets difficulties for high degrees, i.e., problems of 2 variables with degrees 20, 22, and problems of 4, 5 variables with degrees 4, 6, 8 in our examples,
- **raSAT** outperforms **Z3 4.3** in the experiment of three measures. It seems work quite well for high degrees and increasing number of APIs.
- However **raSAT** suffers from enlarging dimensions for both SAT and UNSAT problems. We need further comparison, investigation, and reasonable strategies for increasing dimensions because explosion of boxes occurs when intervals are decomposed many times in **raSAT**.

6.2.2 Experiments on Benchmarks of SMT-LIB

In SMT-LIB [2], benchmarks on non-linear real number arithmetic (the division QF_NRA) are categorized into Meti-Tarski, Keymaera, Kissing, Hong, and Zankl families. Until SMT-COMP 2011, benchmarks are only Zankl family. In SMT-COMP 2012, other families have been added, and currently growing. General comparison among various existing tools on these benchmarks is summarized in Table.1 in [14], which shows Z3 4.3 is one of the strongest.

The brief statistics and explanation of these families are as follows.

- **Meti-Tarski** contains 5364 inequalities among 8377, taken from elementary physics. Typically, they are small problems which have lower degrees and few variables, i.e., 3 or 4 variables in each problem. Linear constraints are frequently mixed in these problems.
- **Keymaera** contains 161 inequalities among 4442.
- **Kissing** has 45 problems, all of which contains equality (mostly single equality).
- **Hong** has 20 inequalities among 20, tuned for QE-CAD and quite artificial.
- **Zankl** has 151 inequalities among 166, taken from termination provers. Problems may contain many (> 100) variables, in which some APIs have > 15 variables

Solver	Hong (20)			Zankl (151)			Meti-Tarski (832)		
	SAT	UNSAT	time(s)	SAT	UNSAT	time(s)	SAT	UNSAT	time(s)
Z3 4.3	0	8	5.620	**50**	**24**	**1144.320**	**502**	**330**	**33.350**
raSAT	0	**20**	381.531	42	9	2417.931	501	156	21.989

Table 6.4: Experimental results for Hong, Zankl, and Meti-Tarski families

Due to restriction on polynomial inequality, we perform experiments only on 20 inequalities of Hong, 151 inequalities of Zankl, and 832 inequalities of Meti-Tarski. Table 6.4 shows the number of solved problems (either SAT or UNSAT) and their total running time (in seconds).

In Hong family, we present interval constraints as $x \in (-\infty, -1) \vee x \in (-1, 1) \vee x \in (1, \infty)$ for each variable x. In Zankl family, all variables of problems are originally given a lower bound which is ≥ 0. We first apply IA to estimate upper and lower bounds of these constraints with an input range $[0, \infty)$ for all variables (note that our implementation of IA allows to estimate infinite bound, i.e., $[0, \infty)$). If IA says IA-SAT (IA cannot decide SAT or UNSAT for the input range $[0, \infty)$), upper bounds for all variables will be manually given and we evaluate almost these problems with a given range $(0, 2)$ (some problems are applied for a range $(0, 4)$). The same is applied for the Meti-Tarski family with a given range $(0, 5)$. For infinite bounds, automatic choices of initial range decomposition will be considered, i.e., $(0, \infty)$ is initially represented as $(0, 2) \vee (2, \infty)$, however it is not yet implemented.

We set the *unit searching* δ to 0.25 and timeout is set to 600 seconds for each problem. Due to trade-off between precision and time consuming, we apply CAI for problems with number of variables ≤ 15, and AF_2 for problems with number of variables > 15.

Among 20 problems of Hong family, their degrees distribute from 1 to 20. **raSAT** solved all of them (all are UNSAT). **Z3 4.3** solved 8 problems, whose degrees are up to 8. **Z3 4.3** becomes timeout for other problems, which have higher degrees than 8. Note that iSAT [12] can solve all of problems in Hong family.

For Zankl family, **Z3 4.3** shows better performance than **raSAT**. **Z3 4.3** runs very fast for problems that contain linear constraints combined with nonlinear constraints of lower degrees (e.g., degree 4). We observed that **raSAT** outperforms **Z3 4.3** for some problems that contain a long monomial (e.g., 60) with higher degrees (e.g., 6) and more

variables (e.g., more than 14). For instance, **raSAT** can solve *matrix-2-all-5,8,11,12* (each less than 36 seconds), while these problem are timeout in **Z3 4.3**, and **raSAT** is quicker to obtain SAT (by testing) in *matrix-2-all-9,10* (3.8 and 2.3 seconds in raSAT; 93 and 436 seconds in Z3 4.3, respectively).

Zankl problems come from termination analysis of term rewriting and are regarded as standard problems for the division QF_NRA of SMT competition. In 2010, the first year of SMT competition for the division QF_NRA, MiniSmt [33] is the winner which can solve 44 among 60 problems (44/60)[1] (43 inequalities and 1 equality). However MiniSmt cannot decide UNSAT problems because of bounded bit encoding. In 2011, the winner Z3 (version 3.1) solved 53/63 problems[2] (38 inequalities and 15 equalities). In 2012, there is no participant reporting comparative results for the division QF_NRA. Recently, there are some proposed approaches for polynomial constraint solving using Zankl problems in comparison with others, i.e., SMT-RAT (22/166) [10], and Z3 4.3 (89/166) [14] in comparison with iSAT (21/166), MiniSmt (46/166), Mathematica (50/166), and QEPCAD (21/166).

Among large number of problems in Meti-Tarski, we extract 832 problems for the experiment. **Z3 4.3** solved all problems, and **raSAT** solved 657 (SAT/UNSAT) problems among 832. Actually, **raSAT** solved almost all SAT problems, but UNSAT problems are less. One reason is that kissing cases occur frequently in UNSAT problems of Meti-Tarski, which **raSAT** cannot handle. Note that, in Table.1 in [14], only QE-CAD based tools work fine (**Z3 3.1** does not apply QE-CAD, whereas **Z3 4.3** = **nlSAT** includes QE-CAD). Although **raSAT** has certain limitations on UNSAT problems, it shows enough comparable results in Meti-Tarski benchmarks and seems faster than most of QE-CAD based tools (except for **Z3 4.3**).

Observation

At the moment, **raSAT** is not strong as **Z3 4.3** in Zankl family. However we feel that the results are encouraging. We judge that results of **raSAT** would be improved by solving linear fragment separated from non-linear constraints.

- The linear fragment indicates vertices of intersecting, which would be candidates of

[1] http://www.smtcomp.org/2010/
[2] http://www.smtcomp.org/2011/

cutoff points for guiding interval decomposition.

- Solving the linear fragment can help to prove UNSAT directly or quicker by narrowing domains of searching, i.e., identifying UNSAT domains. For instance, the linear constraints $x_1 - x_2 > 0 \wedge x_2 - x_1 > 0$ are contained in a problem of Zankl family but **raSAT** cannot prove it UNSAT by interval arithmetic and interval decomposition.

Chapter 7

Extensions to Polynomial Equality

We extend our approach for polynomial constraints including greater-than-or-equal constraints and equality constraints.

7.1 Greater-Than-or-Equal Handling

We first give the definition of **strict-UNSAT** for greater-than-or-equal constraints ($\geq$).

Definition 7.1.1. The constraint $\bigwedge_j f_j \geq 0$ is said **strict-UNSAT** if $\bigwedge_j f_j > -\delta_j$ is UNSAT for a $\delta_j > 0$.

Similar for IA-strict-UNSAT, $f > 0$ is said **IA-strict-UNSAT** if $f > -\delta$ is IA-UNSAT for $\delta > 0$. We have two lemmas as followings.

Lemma 7.1.2. *If the constraint $\bigwedge_j f_j \geq 0$ is **strict-UNSAT**, it is really UNSAT.*

Lemma 7.1.3. *If the constraint $\bigwedge_j f_j > 0$ is **SAT**, then the constraint $\bigwedge_j f_j \geq 0$ is **SAT**.*

For the greater-than-or-equal constraints $\bigwedge_j f_j \geq 0$, we apply Lemma 7.1.2 for proving UNSAT, and Lemma 7.1.3 for proving SAT. Note that if $\bigwedge_j f_j \geq 0$ is SAT but $\bigwedge_j f_j > 0$ is UNSAT (i.e., kissing situation), **raSAT** says *unknown* and when $\bigwedge_j f_j = 0$ is represented by $\bigwedge_j (f_j \geq 0 \wedge f_j \leq 0)$, **raSAT** simply answers *unknown*. In general, strict-UNSAT is only applied for greater-than-or-equal constraints ($f_j \geq 0$) and IA-UNSAT is replaced by IA-strict-UNSAT.

7.2 Polynomial Equality Handling

7.2.1 Polynomial Equality by Intermediate Value Theorem

For solving polynomial constraints with an equality ($g = 0$), we apply intermediate value theorem. That is, if existing 2 test cases such that $g > 0$ and $g < 0$ then $g = 0$ is SAT.

Lemma 7.2.1. *For a polynomial constraint*

$$F = (x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)) \bigwedge_j^m f_j(x_1, \cdots, x_n) > 0 \ \wedge \ g(x_1, \cdots, x_n) = 0,$$

if

(i) existing a box $(l_1, h_1) \times \cdots \times (l_n, h_n)$ *such that* $(l_i, h_i) \subseteq (a_i, b_i)$ *and* $\bigwedge_j^m f_j(x_1, \cdots, x_n) > 0$ *is IA-VALID in the box,*

(ii) existing two instances $(t_1, \cdots, t_n), (t'_1, \cdots, t'_n)$ *in the box such that* $g(t_1, \cdots, t_n) > 0$ *and* $g(t'_1, \cdots, t'_n) < 0$,

then F is SAT.

We apply Lemma 7.2.1 for proving SAT of polynomial constraints containing an equality. In implementation, we first find an IA-VALID box by interval decomposition and interval arithmetic for inequality constraints (i), and then find 2 instances in the IA-VALID box by testing (ii). Note that, for (i), testing can guide for finding IA-VALID boxes, i.e., a small box containing a test data which is Test-SAT for inequalities. Figure 7.1 demonstrates our approach when applying intermediate value theorem.

In Table 7.1 we show preliminary experiment for 15 problems that contain polynomial equalities in Zankl family. **raSAT** works well for these SAT problems and it can detect all SAT problems (11 among 15). At the current implementation, **raSAT** reports *unknown* for UNSAT problems. The first 4 columns indicate *name of problems*, *the number of variables*, *the number of polynomial equalities*, and *the number of inequalities* in each problem, respectively. The last 2 columns show comparison results of **Z3 4.3** and **raSAT**.

We also apply the same idea for multiple equalities $\bigwedge_i g_i = 0$ such that $Var(g_k) \cap Var(g_{k'}) = \emptyset$, where $Var(g_k)$ is denoted for the set of variables in the polynomial g_k, e.g., *gen-05* to *gen-09* in Table 7.1. In the next section we will present idea for solving general cases of multiple equalities.

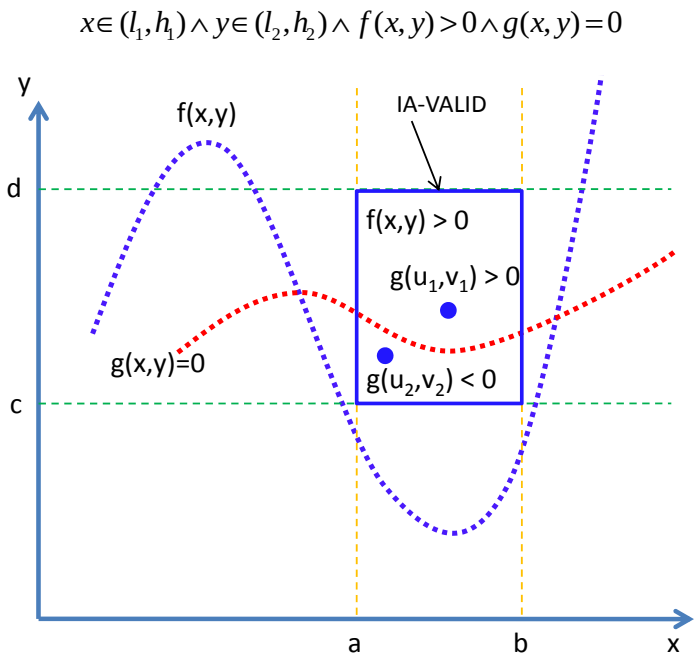

Figure 7.1: Solving equality by intermediate value theorem

Problem Name	No. Variables	No. Equalities	No. Inequalities	**Z3 4.3** (15/15)		**raSAT** (11/15)	
				Result	Time(s)	Result	Time(s)
gen-03	1	1	0	SAT	0.01	SAT	0.015
gen-04	1	1	0	SAT	0.01	SAT	0.015
gen-05	2	2	0	SAT	0.01	SAT	0.046
gen-06	2	2	1	SAT	0.01	SAT	0.062
gen-07	2	2	0	SAT	0.01	SAT	0.062
gen-08	2	2	1	SAT	0.01	SAT	0.062
gen-09	2	2	1	SAT	0.03	SAT	0.062
gen-10	1	1	0	SAT	0.02	SAT	0.031
gen-13	1	1	0	UNSAT	0.05	unknown	0.015
gen-14	1	1	0	UNSAT	0.01	unknown	0.015
gen-15	2	3	0	UNSAT	0.01	unknown	0.015
gen-16	2	2	1	SAT	0.01	SAT	0.062
gen-17	2	3	0	UNSAT	0.01	unknown	0.031
gen-18	2	2	1	SAT	0.01	SAT	0.078
gen-19	2	2	1	SAT	0.05	SAT	0.046

Table 7.1: Experimental results for 15 equality problems of Zankl family

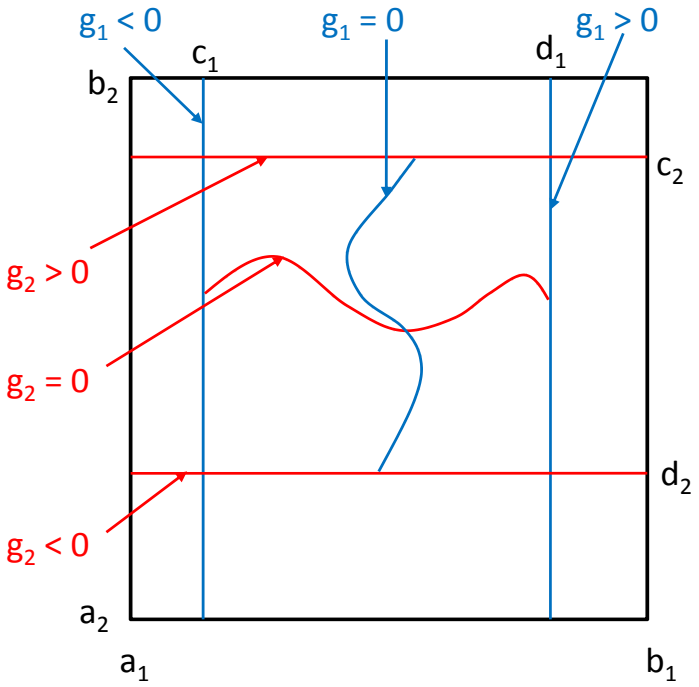

Figure 7.2: Solving multiple equalities

7.2.2 Extensions to Multiple Equalities

We first present an approach for solving polynomial constraints with 2 equalities and then extend the same idea for general cases. We assume that constraints have two variables.

For a polynomial constraint $F = (x_1 \in (a_1, b_1) \wedge x_2 \in (a_2, b_2)) \bigwedge_j f_j > 0 \wedge g_1 = 0 \wedge g_2 = 0$, F can be proved as SAT by following steps:

- first, find a box such that $\bigwedge_j f_j > 0$ is IA-VALID, i.e., $(l_1, h_1) \times (l_2, h_2)$,
- find 2 instances $c_1, d_1 \in (l_1, h_1)$ such that $g_1 < 0$ on $\{c_1\} \times (l_2, h_2)$ and $g_1 > 0$ on $\{d_1\} \times (l_2, h_2)$ (values of g_1 are estimated by interval arithmetic),
- find 2 instances $c_2, d_2 \in (l_2, h_2)$ such that $g_2 < 0$ on $(l_1, h_1) \times \{c_2\}$ and $g_2 > 0$ on $(l_1, h_1) \times \{d_2\}$ (values of g_2 are also estimated by interval arithmetic).

Figure 7.2 demonstrates our approach when solving polynomial constraints with 2 equalities. The idea behind is finding a box such that IA-VALID for polynomial inequalities and proving that the line $g_1 = 0$ intersects the line $g_2 = 0$ insides the IA-VALID box.

The idea above is applied for multiple equalities. For a polynomial constraint $F = (x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)) \bigwedge_j f_j > 0 \bigwedge_{k=1}^{m} g_k = 0$, F can be proved as SAT by following steps:

- first, find a box such that $\bigwedge_j f_j > 0$ is IA-VALID, i.e., $(a_1, b_1) \times \cdots \times (a_n, b_n)$,
- find m instances $c_1, \cdots, c_m$ in the IA-VALID box such that $g_i < 0$ is IA-VALID with $(a_1, b_1) \times \cdots \{c_i\} \times \cdots \times (a_n, b_n)$ (bounds of g_i are estimated by interval arithmetic),
- find m instances $d_1, \cdots, d_m$ in the IA-VALID box such that $g_i > 0$ is IA-VALID with $(a_1, b_1) \times \cdots \{d_i\} \times \cdots \times (a_n, b_n)$ (bounds of g_i are estimated by interval arithmetic),

with a restriction on variables of equalities $\forall i_1, \cdots, i_k. \mid \cup_{1 \leq l \leq k} Var(g_{il}) \mid \geq k$. The restriction guarantees that lines $g_k = 0$ have the same intersection point in the IA-VALID box.

For proving UNSAT of polynomial equality, we transform equalities into inequality forms, i.e., for some $\delta > 0$, $\bigwedge_{k=1}^{m} g_k = 0$ are transformed into $\bigwedge_{k=1}^{m} -\delta < g_k < \delta$. We apply Lemma 7.2.2 for proving UNSAT of polynomial inequalities.

Lemma 7.2.2. *For a polynomial constraint* $F = (x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)) \bigwedge_j f_j > 0 \bigwedge_{k=1}^{m} g_k = 0$, *if* $F' = (x_1 \in (a_1, b_1) \wedge \cdots \wedge x_n \in (a_n, b_n)) \bigwedge_j f_j > 0 \bigwedge_{k=1}^{m} (-\delta < g_k < \delta)$ *is UNSAT for some* $\delta > 0$, *then* F *is UNSAT.*

Chapter 8

Conclusions

8.1 Summary of the Thesis

The aim of research is to propose an approach and develop an SMT for solving polynomial constraints. In the thesis, we propose an iterative approximation refinement scheme, which is applied for solving polynomial constraints on real numbers. The approximation scheme consists of *interval arithmetic* (IA) (over-approximation, aiming to decide unsatisfiability), and *testing* (under-approximation, aiming to decide satisfiability). If both of them fail to decide, *refinement* is applied that allows to focus searching on narrower domains. This scheme is described as an abstract DPLL(T) procedure, which performs as a refinement loop of over approximation (IA), under approximation (testing), and refinement guided from IA and testing. We implemented the scheme as the SMT solver **raSAT**.

Comparison with the SMT solver **Z3 4.3**, believed as the strongest SMT solver for non-linear real arithmetic, was performed by preliminary evaluation on the different measures (i.e., high degrees, number of variables, number of APIs), and experiments on the benchmarks of SMT-LIB. In comparison with **Z3 4.3**, **raSAT** is a little weak for *Zankl* problems, however **raSAT** worked beyond **Z3 4.3** in the preliminary evaluation and for the *Hong* problems, and **raSAT** shows enough comparable results in Meti-Tarski problems. By applying single method without tuning in implementation, the experimental results of **raSAT** seem encouraging and **raSAT** is still in improvements.

Our contributions are summarized as follows:

- We propose an approach of *iterative approximation refinement* for solving con-

straints, which is formalized as an *abstract DPLL(T) procedure* for *over/under-approximations* and *refinements* under a background theory T in Chapter 3. An under approximation is sound for proving in the background theory T, and an over approximation is sound for disproving. When they neither prove nor disprove, *refinements* are applied to decompose an atomic formula of the input formula, i.e., ψ to $\psi_1 \vee \psi_2$ such that $\psi \Leftrightarrow \psi_1 \vee \psi_2$. The proposed approach combined DPLL(T) procedure with over/under-approximations and refinements is sound and complete for solving polynomial inequality constraints under certain restrictions.

- In Chapter 4, we instantiate *interval arithmetic* as an over approximation and *testing* as an under approximation. A variant of affine interval, called *Chebyshev Affine Interval* (CAI), is newly proposed. CAI has an advantage over existing ones (e.g., AF, AF_1, AF_2) such that it keeps sources of computation for high degree variables (i.e., by introducing $|\epsilon|$). This would be useful for guiding refinements from influential variables.

- In Chapter 5, we propose UNSAT cores of polynomial constraints that can improve efficiency in theory propagation, i.e., the number of clauses for learning is reduced. Computation of UNSAT cores in polynomial constraints allows inferring other UNSAT domain when a particular domain is detected as UNSAT. Though current implementation for UNSAT core is not well organized and we need further improvement. When performing a large number of test data (i.e., a large number of variables), the proposed approach for incremental test data generation would be useful.

- One of key strategies is the choice of intervals to decompose into smaller intervals. Otherwise, interval decompositions lead exponentially many boxes with respect to the number of variables. The proposed strategy for selecting intervals showed effective results. Such strategy consists of three steps. First, after dynamically sorting polynomials appearing in constraints with respect to certain dependencies, we concentrate on the first polynomial constraint that testing cannot find a SAT instance. Second, choice on variables appearing in that polynomial is based on *sensitivity*, which is detected during previous interval arithmetic computation. Finally, after

variables for interval decompositions are selected, a method for decomposing an interval is chosen, such as *monotonic decomposition* or *tick decomposition*.

- Polynomial inequality handling is extended for *greater-than-or-equal* ($\geq$) constraints, i.e., $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > 0$ for proving SAT, and for proving UNSAT $\bigwedge_i f_i \geq 0$ is transformed to $\bigwedge_i f_i > -\delta_i$ for $\delta_i > 0$.
- Solving polynomial constraints including equalities is extended in Chapter 7 by a non-constructive approach based on *intermediate value theorem*. Preliminary experiments of **raSAT** for polynomial equalities show that the non-constructive approach reasonably works and its results seem to be comparable with **Z3 4.3** (i.e., 11 solved problems among 15 equalities of Zankl family).

8.2 Future Directions

We are just in the beginning and have lots of future work in both development of **raSAT** and its extensions.

8.2.1 raSAT Development

- **Avoiding local optimality**: we plan to borrow the similar idea of *restart* in MiniSAT for escaping from hopeless local search (i.e., solution set is not dense or empty). *Heuristics* would be, after a deep interval decomposition of a box and Test-UNSAT are reported, backtrack occurs to choose a randomly selected box.
- **Separation of linear constraints**: Many benchmarks contain linear constraints. We expect improvement by separating them for existing SMT with Presburger arithmetic. From their results, vertices of intersecting linear constraints would be candidates of cutoff points for interval decompositions.

  For example, if two linear constraints $y-2x+1 > 0$ and $y+x-2 > 0$ are included in a polynomial inequality constraint, their intersecting vertices $x = 1, y = 1$ are applied for interval decomposition. Assume that the intervals $x \in (0, 5)$ and $y \in (-10, 2)$

are used for interval decomposition, choices are,

$$x \in (0,5) \quad \Leftrightarrow x \in (0,1) \vee x \in (1,5)$$
$$y \in (-10,2) \quad \Leftrightarrow y \in (-10,1) \vee y \in (1,2)$$

- **Incremental DPLL**: For interactions with a SAT solver, we currently apply the very lazy theory learning. Combination of *less lazy* and *eager* theory propagation would improve efficiency, in which we can propagate a conflict from a partial truth assignment instead of waiting for a full truth assignment obtained by the SAT solver.

- **Error guaranteed floating point arithmetic**: Currently, standard floating point arithmetic is used for both interval arithmetic and testing. Although this has benefit on comparison between floating and fix point arithmetic behavior (e.g., round-off/overflow error analysis [23]), the result may not be sound for exact real arithmetic. Since polynomial inequality permits the reduction from real numbers to rational numbers, we are planning to apply exact rational number arithmetic packages, e.g., the numerical packages of Ocaml.

- **Other directions**: we need further investigation for refinement strategies (both strategies and experiments), i.e., choice of variables for interval decomposition, how to decompose an interval into smaller intervals (i.e., based on sensitivity of variables), etc.

8.2.2 Extensions of raSAT Loop

- **Solving polynomial constraints on integer numbers**: we can extend our approach for integer numbers. In this domain, number of test data is finite if interval constraints are bounded and Test-UNSAT can imply UNSAT if all test data is generated. A tight interaction between testing, i.e., generating test data, testing results, and interval decomposition could be investigated.

- **Equality handling**: currently, **raSAT** loop can handle only an inequality by *intermediate value theorem*. We need further investigations, such as giving formal proof of our idea for multiple equalities and showing experimental results. Com-

bining ideal based technique for handling equality, such as *Gröbner basis*, is also considered.

8.3 Applications

We plan to apply our SMT solver **raSAT** for some problems of software/hardware verification, loop invariant generation, such as,

- **Checking overflow and roundoff error**: In the computers, the real numbers are represented by finite numbers (i.e., floating point numbers, fixed point numbers). Due to finite representation, the over-flow and roundoff errors (OREs) may occur. The OREs will be propagated through computations of the program. Further, the computations themselves also cause OREs because the arithmetic needs to round the result to fit the number format. Besides, OREs are also affected by types of statements, i.e., branch, loop, assignment statements. By symbolic execution, ORE constraints are propagated from a program and ORE problems are reduced to problems of solving ORE constraints for verifying whether OREs occur.

- **Loop invariant generation**. The use of Farkas's lemma is a popular approach in linear loop invariant generation [7]. Farkas's lemma uses products of matrices, and it requires solving polynomial constraints of degree 2. Non-linear loop invariant generation [28] and hybrid systems [29] require more complex polynomials. We can extend the target for *non-linear loop invariant generation*

Bibliography

[1] Anai, H. Algebraic methods for solving real polynomial constraints and their applications in biology. In *In Algebraic Biology – Computer Algebra in Biology* (2005), pp. 139–147.

[2] Barrett, C., Stump, A., and Tinelli, C. The satisfiability modulo theories library (SMT-LIB).

[3] Borralleras, C., Lucas, S., Navarro-Marset, R., Rodríguez-Carbonell, E., and Rubio, A. Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In *Proceedings of the 22nd International Conference on Automated Deduction* (2009), CADE-22, Springer-Verlag, pp. 294–305.

[4] Bourbaki, N. *General Topology*. Springer-Verlag, 1989.

[5] Bryant, R. E., Kroening, D., Ouaknine, J., Seshia, S. A., Strichman, O., and Brady, B. Deciding bit-vector arithmetic with abstraction. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems* (2007), TACAS'07, Springer-Verlag, pp. 358–372.

[6] Collins, G. E. Quantifier elimination by cylindrical algebraic decomposition – twenty years of progress. In *Quantifier Elimination and Cylindrical Algebraic Decomposition* (1998), B. F. Caviness and J. R. Johnson, Eds., Springer-Verlag, pp. 8–23.

[7] Colón, M., Sankaranarayanan, S., and Sipma, H. Linear invariant generation using non-linear constraint solving. In *CAV* (2003), vol. 2725 of *Lecture Notes in Computer Science*, Springer, pp. 420–432.

[8] Comba, J. L. D., and Stolfi, J. Affine arithmetic and its applications to computer graphics. In *Proceedings of VI SIBGRAPI.* (1993), pp. 9–18.

[9] Corzilius, F., and Ábrahám, E. Virtual substitution for SMT-solving. In *Proceedings of the 18th international conference on Fundamentals of computation theory* (2011), FCT'11, Springer-Verlag, pp. 360–371.

[10] Corzilius, F., Loup, U., Junges, S., and Ábrahám, E. SMT-RAT: an SMT-compliant nonlinear real arithmetic toolbox. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing* (2012), SAT'12, Springer-Verlag, pp. 442–448.

[11] Daniel Kroening, O. S. *Decision Procedures: An Algorithmic Point of View*. Springer-Verlag, 2008.

[12] Franzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation 1* (2007), 209–236.

[13] Ganai, M., and Ivancic, F. Efficient decision procedure for non-linear arithmetic constraints using cordic. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009* (2009), pp. 61 –68.

[14] Jovanović, D., and de Moura, L. Solving non-linear arithmetic. In *Proceedings of the 6th international joint conference on Automated Reasoning* (2012), IJCAR'12, Springer-Verlag, pp. 339–354.

[15] Khanh, T. V., and Ogawa, M. SMT for polynomial constraints on real numbers. *Electr. Notes Theor. Comput. Sci. 289* (2012), 27–40.

[16] Klaas Pieter Hart, J.-i. N., and Vaughan, J. E. *Encyclopedia of General Topology*. Elsevier, 2003.

[17] Lucas, S., and Navarro-Marset, R. Comparing csp and sat solvers for polynomial constraints in termination provers. *Electron. Notes Theor. Comput. Sci. 206* (Apr. 2008), 75–90.

[18] Messine, F. Extensions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science 8*, 2 (2002).

[19] Microsoft. Z3.

[20] Miyajima, S., T. M., and Kashiwagi, M. A new dividing method in affine arithmetic. *IEICE Transactions E86-A*, 9 (2003), 2192–2196.

[21] Moore, R. E. *Interval Analysis*. Prentice-Hall, 1966.

[22] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference* (2001), DAC '01, ACM, pp. 530–535.

[23] Ngoc, D. T. B., and Ogawa, M. Overflow and roundoff error analysis via model checking. In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods* (2009), SEFM '09, IEEE Computer Society, pp. 105–114.

[24] Ngoc, D. T. B., and Ogawa, M. Checking roundoff errors using counterexample-guided narrowing. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010), ASE '10, ACM, pp. 301–304.

[25] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Abstract DPLL and abstract DPLL modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, F. Baader and A. Voronkov, Eds., vol. 3452 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005, pp. 36–50.

[26] Passmore, G. O., and Jackson, P. B. Combined decision techniques for the existential theory of the reals. In *Proceedings of the 16th Symposium, 8th International Conference. Held as Part of CICM '09 on Intelligent Computer Mathematics* (2009), Calculemus '09/MKM '09, Springer-Verlag, pp. 122–137.

[27] Ratschan, S. Efficient solving of quantified inequality constraints over the real numbers. *ACM Trans. Comput. Logic 7*, 4 (Oct. 2006), 723–748.

[28] Sankaranarayanan, S., Sipma, H. B., and Manna, Z. Non-linear loop invariant generation using gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2004), POPL '04, ACM, pp. 318–329.

[29] Sankaranarayanan, S., Sipma, H. B., and Manna, Z. Constructing invariants for hybrid systems. *Form. Methods Syst. Des. 32*, 1 (2008), 25–55.

[30] Stolfi, J. *Self-Validated Numerical Methods and Applications.* PhD thesis, PhD. Dissertation, Computer Science Department, Stanford University, 1997.

[31] Tarski, A. A decision method for elementary algebra and geometry. *Bulletin of the American Mathematical Society 59* (1951).

[32] Weispfenning, V. Quantifier elimination for real algebra - the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing 8* (1997), 85–101.

[33] Zankl, H., and Middeldorp, A. Satisfiability of non-linear (ir)rational arithmetic. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning* (2010), LPAR'10, Springer-Verlag, pp. 481–500.

Publications

[1] To Van Khanh, Mizuhito Ogawa. *SMT for Polynomial Constraints on Real Numbers.* (TAPAS' 2012) Electronic Notes in Theoretical Computer Science, vol. 289, Dec 2012, pages 27 – 40.

[2] To Van Khanh, Mizuhito Ogawa. *raSAT: SMT for Polynomial Inequality.* JAIST Technical Report 2013, IS-RR-2013-003, `http://hdl.handle.net/10119/11349`.