Improving NLSAT for Nonlinear Real Arithmetic

Zhonghan Wang*†
*Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
wangzh@ios.ac.cn, wangzhonghan272@gmail.com

***Abstract*—The Model-Constructing Satisfiability Calculus (MCSAT) framework has been applied to SMT problems over various arithmetic theories. NLSAT, an implementation using cylindrical algebraic decomposition (CAD) for explanation, is especially competitive for nonlinear real arithmetic (NRA) constraints. However, current Conflict-Driven Clause Learning (CDCL)-style algorithms only consider literal information when making decisions, and thus ignore the influence of clauses on arithmetic variables. This limitation may lead NLSAT to encounter unnecessary conflicts due to suboptimal literal choices. To address this issue, we analyze conflicts caused by literal decisions and incorporate clause-level information that directly affects arithmetic variables. We propose two main algorithmic improvements: a clause-level feasible-set-based look-ahead mechanism and an arithmetic propagation-based branching heuristic. We implement our solver, named clauseSMT, based on a dynamic variable ordering framework. Experiments indicate that clauseSMT is competitive on nonlinear real arithmetic problems compared with existing SMT solvers (CVC5, Z3, YICES2), and it outperforms all of them on satisfiable instances of SMT(QF_NRA) in SMT-LIB. We also evaluate the effectiveness of our proposed methods.**

***Index Terms*—NLSAT, nonlinear real arithmetic, SMT, clause level.**

I. Introduction

A. Motivation

Satisfiability Modulo Theories (SMT) refers to the problem of determining the satisfiability of formulas in first-order logic. SMT problems typically involve theories such as linear and nonlinear arithmetic, uninterpreted functions, strings, and arrays [1]. As a fundamental problem in software engineering, formal methods, and programming languages, SMT has widespread applications, including symbolic execution [2], [3], program verification [4], [5], program synthesis [6], automata learning [7], [8], and neural network verification [9]–[12].

Nonlinear real arithmetic (NRA) is a class of arithmetic theories. It consists of atoms represented as inequalities over polynomials, and is therefore sometimes referred to as the theory of polynomial constraints. Variables can take Boolean or real values, depending on their types. SMT(NRA) instances are typically generated from both academic and industrial applications. They are commonly used in cyber-physical systems [13]–[15], ranking function generation [16], [17], and nonlinear hybrid automata analysis [18]. Instances from these applications are collected in the SMT-LIB benchmarks [19]. The high performance of SMT solvers over nonlinear arithmetic has significantly improved these applications.

Decision procedures for solving nonlinear arithmetic are usually based on cylindrical algebraic decomposition (CAD) [20], a widely used tool for real quantifier elimination. CAD generates the current unsatisfiable cell during the search procedure and is employed in modern algorithms. Among these, NLSAT [21] is a mainstream algorithm that leverages CAD for lemma generation. Its core idea is to assign values directly to arithmetic variables, rather than at the literal level as in CDCL(T).

Although NLSAT introduces the novel approach of directly assigning arithmetic variables, it still relies on literal decisions when processing arithmetic clauses containing several unevaluated literals. These decided literals are then used for conflict analysis within the CDCL-style framework. However, improper literal decisions can sometimes induce conflicts, slowing down the overall search process. Therefore, a heuristic for literal decisions is necessary.

We identify three central problems and present our solutions in the context of algorithmic improvements:

- What factors cause conflicts in the NLSAT algorithm, and can some of them be avoided?
- Is it possible to assign values directly to arithmetic variables, independent of literal-level decision information, within a CDCL-style framework?
- Can propagation be performed on arithmetic variables, analogous to unit propagation in SAT solving, and is this new propagation method effective for guiding assignments and detecting conflicts?

B. Contributions

To address the questions above, this paper proposes, for the first time, a new algorithm that incorporates clause-level information.

First, we analyze the conflict problems that arise in NLSAT and categorize them into two types. As described in [22], each clause narrows the feasible set[1] of an arithmetic variable. This technique has previously been used to enlarge the operation choices in local search algorithms, but it has not been considered in complete methods like NLSAT. Consequently,

This work was conducted during the author's Master studies at the Institute of Software, Chinese Academy of Sciences.

[1]Also called the satisfying domain in [22].

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00128

arithmetic variables can sometimes be narrowed to an empty search space, causing conflicts. We describe this type of problem from the perspective of interval arithmetic and propose a solution based on the computation of feasible intervals. The clause-level feasible-set idea extends the spirit of NLSAT by directly guiding assignments to arithmetic variables.

Second, we introduce an incremental computation of the clause-level feasible set, followed by the definition of **clause-level propagation**. In SAT solvers, unit propagation is an effective tool to deduce assignments and detect conflict clauses. Analogously, clause-level propagation is employed to fix a possible witness for an arithmetic variable or to quickly detect empty feasible-set cases.

Finally, we present the structure and implementation details of our solver, `clauseSMT`. Although dynamic variable ordering has been discussed in [23], SMT-RAT [24] solves fewer instances due to the lack of efficient data structures. We present techniques and data structures inspired by SAT solving. Our implementation extends the NLSAT module of the Z3 solver [25], relying on existing libraries in Z3 for mathematical operations such as root isolation, polynomial operations, and algebraic number representation. Experiments on the SMT-LIB benchmark demonstrate the effectiveness of our proposed techniques, including the look-ahead mechanism and clause-level propagation. The results show that `clauseSMT` solves the most satisfiable instances and is highly competitive against other SMT solvers overall.

In summary, this paper makes the following contributions:

- We propose a new MCSAT-based method for nonlinear arithmetic, introducing **clause-level feasible sets** to avoid conflicts caused by literal-level decisions.
- We define **clause-level propagation**, which quickly detects conflict cases or fixes values for arithmetic variables.
- We integrate the propagation method into the VSIDS branching heuristic, guiding the search process and reducing semantic stages.
- We implement these ideas in our solver `clauseSMT` and conduct experiments on SMT-LIB benchmarks to demonstrate the effectiveness of our approach.

C. Structure of the Paper

The paper is organized as follows. In Section II, we introduce SMT problems over nonlinear real arithmetic and review the traditional complete method NLSAT. Section III analyzes the conflicts that occur in the NLSAT algorithm. In Section IV, we present a feasible-set based look-ahead mechanism. Building on the concept of clause-level information, Section V introduces the clause-level propagation algorithm and a new branching heuristic. Section VI discusses the details of implementation. We compare our solver with other SMT solvers and perform an ablation study in Section VII. Related work on solving non-linear real arithmetic is reviewed in Section VIII. Finally, Section IX concludes the paper and outlines potential directions for future research.

D. Artifact Availability

To facilitate reproducibility and further research, we release the full implementation of ClauseSMT, together with all experimental data and scripts, as an open-source artifact. The artifact is publicly available on a GitHub repository[2], enabling researchers to replicate our experiments and investigate the solver's performance on the QF_NRA benchmark.

II. Preliminaries

This section introduces the basic definitions of SMT problems over nonlinear real arithmetic, followed by a review of the NLSAT algorithm. In addition, we present the computation of clause-level feasible sets.

A. Syntax of SMT(QF_NRA)

The syntax of SMT constraints over nonlinear real arithmetic is defined as follows:

$$\begin{aligned}
\text{arithmetic variables:} &\quad x \in \mathbb{V} \\
\text{boolean variables:} &\quad b \in \mathbb{B} \\
\text{polynomials:} &\quad p := x \mid c \mid p + p \mid p \cdot p \\
\text{atoms:} &\quad a := b \mid p \leq 0 \mid p \geq 0 \mid p = 0 \\
\text{literals:} &\quad l := a \mid \neg a \\
\text{formulas:} &\quad \varphi := l \mid \varphi \vee \varphi \mid \varphi \wedge \varphi
\end{aligned}$$

An atom is either a Boolean atom, defined by a Boolean variable $b \in \mathbb{B}$, or an arithmetic atom, defined by a (non-strict) inequality or equality of a polynomial over $\mathbb{V}$. A literal is either an atom or its negation. A clause is a disjunction of literals, and all input formulas are transformed into conjunctive normal form (CNF), i.e., a conjunction of clauses. SMT(NRA) refers to the set of formulas over the theory of nonlinear real arithmetic.

For the semantics, we define an *assignment* α as a mapping from variables to values.

- A *Boolean assignment* maps Boolean variables to truth values, denoted as $\alpha_{\mathsf{bool}} : b \mapsto \{\top, \bot\}$.
- An *arithmetic assignment* maps real variables to real numbers, denoted as $\alpha_{\mathsf{real}} : x \mapsto \mathbb{R}$.

A *full assignment* maps all Boolean and real variables, while a *partial assignment* only covers a subset. Under a given assignment, each atom is evaluated as follows:

1) *true*, if the assignment satisfies it;
2) *false*, if the assignment violates it;
3) *undefined or unevaluated*, if it contains variables not yet assigned.

A full assignment that makes all clauses true is called a *model* (or *solution*) of the formula, certifying its satisfiability. The SMT(QF_NRA) problem is to decide whether such a model exists for a given input formula, or to prove that none does.

[2]https://github.com/yogurt-shadow/ClauseSMT_ASE2025

B. Feasible Set

For nonlinear real arithmetic constraints, a key technique for determining the possible values of arithmetic variables is *root isolation*. Given an arithmetic atom of the form

$$p \ \{\leq, \geq, =, >, <\} \ 0,$$

if exactly one variable v remains unassigned under the current assignment, we can compute the set of values of v that satisfy the atom. We call this set the *feasible set*.

For higher-order polynomial constraints, feasible sets are usually computed via root isolation, which determines the roots of the polynomial. These roots partition the real line into intervals[3]. Each interval preserves a fixed truth value of the atom, and the union of all satisfying intervals forms the overall feasible set.

For negated literals, the feasible set is simply the complement of the feasible set of the corresponding positive atom. By restricting the variable v to any value in the feasible set, the atom is guaranteed to be satisfied[4].

Besides the literal level, the notion of a feasible set can also be extended to the clause level, where it represents the set of values that make the entire clause satisfied. In this work, we focus on the case where exactly one arithmetic variable in the clause is left unassigned; we call such a clause a *univariate clause*.

Definition 1 (Feasible Set)**.** *Let l be a literal, x an arithmetic variable, and α an assignment that maps all variables in l except x. The* feasible set *(resp.* infeasible set*) of l is the union of intervals over $\mathbb{R}$ such that l is satisfied (resp. unsatisfied) when x is assigned any value from the interval.*

Similarly, let c be a clause, x an arithmetic variable, and α an assignment that maps all variables in c except x. The feasible set *(resp.* infeasible set*) of c is the set of all values of x that make c satisfied (resp. unsatisfied). It can be computed by taking the union (resp. intersection) of the feasible sets (resp. infeasible sets) of all literals in c.*

Example 1 illustrates the construction of a feasible set.

Example 1. *Consider an assignment $\alpha := \{b \mapsto \bot, x \mapsto 0\}$. The feasible set of the clause*

$$b \vee (y + x > 0) \vee (y^2 > 2)$$

is

$$(-\infty, -\sqrt{2}) \ \cup \ (0, \infty),$$

since these are the values of y that satisfy the clause given the current assignment.

C. Original NLSAT Algorithm

NLSAT is the core algorithm for nonlinear real arithmetic (NRA) within the Z3 solver [25]. It handles both boolean and arithmetic variables directly, integrating theory reasoning into the CDCL framework. Specifically, NLSAT extends traditional unit propagation and boolean decisions to real-variable propagation (R-propagation) and semantic decisions.

To select an appropriate value for an arithmetic variable, NLSAT incrementally updates its feasible-set during the search. Let the current feasible-set be $curr\_set$, the feasible-set of a literal lit be lit_set, and real-variable propagation take effect under the following circumstances:

- **lit_set is empty**: the literal is propagated as false, since no value can satisfy it.
- **lit_set is full**: the literal is propagated as true, since any value satisfies it.
- **curr_set is a subset of lit_set**: the literal is propagated as true, because the current feasible-set already satisfies it.
- **curr_set has no intersection with lit_set**: the literal is propagated as false, because all values in the current feasible-set violate the literal.

When processing a clause containing both boolean and arithmetic variables, NLSAT first applies propagation and evaluation to detect evaluated literals. If one literal is true, the clause is skipped; otherwise, NLSAT decides the first unevaluated literal and updates the feasible-set accordingly. Algorithm 1 presents the detailed processing steps in NLSAT.

For conflict analysis, NLSAT employs cylindrical algebraic decomposition (CAD) as an explanation tool. Using model-based projection, it identifies the conflict cell and generates a lemma to prevent the solver from revisiting the same conflict in the future. Algorithm 2 presents the complete NLSAT procedure.

III. Conflicts During the NLSAT Algorithm

In this section, we analyze the sources of conflicts in NLSAT algorithms. Broadly, these conflicts can be categorized into two types: those caused by semantic decisions and those caused by literal decisions.

A. Conflicts Caused by Semantic Decisions

In SAT solving, conflicts are typically caused by literal (i.e., boolean variable) decisions. For a satisfiable instance, a SAT solver can avoid conflicts if it uses a perfect phase selection strategy (i.e., assigns variables correctly to true or false). For unsatisfiable instances, conflicts are unavoidable regardless of the phase choices. In both cases, when CDCL detects a conflict due to incorrect decision values, conflict analysis is invoked to generate a new lemma that forces a change in the previous assignment.

Similarly, in the NLSAT algorithm for SMT solving, conflicts can arise from incorrect semantic decisions, i.e., selecting a value from a given interval. As discussed in [26], the search space of nonlinear arithmetic is partitioned into sign-invariant cells. However, in systematic solvers like NLSAT, the current cell being explored cannot be predicted in advance, and therefore conflicts may occur. For unsatisfiable instances, every cell in the search space is inconsistent with at least

[3]In the terminology of cylindrical algebraic decomposition, such intervals are also called *cells*.

[4]The feasible set may also be empty or cover the entire real line, meaning the atom is always unsatisfiable or always satisfied, respectively.

Algorithm 1: Clause Processing in NLSAT

Input : A set of clauses F
Output: Conflict clause $conf\_cls$, or `No Conflict`

for *each clause* $c \in F$ **do**
  for *each literal* $l \in c$ **do**
    $lit\_set \leftarrow$ compute feasible-set of l;
    $val \leftarrow$ real propagate l using lit_set;
    if $val = \top$ **then**
      break;
      `// clause is satisfied, skip remaining literals`
    if $val = \bot$ **then**
      continue;
      `// literal unsatisfied, check next literal`
  if *exist satisfied literal in* c **then**
    continue;
    `// clause satisfied, check next clause`
  else if *exactly one literal undefined in* c **then**
    unit propagate the literal;
  else if *two or more literals undefined in* c **then**
    decide the first undefined literal;
  else
    `// all literals are unsatisfied`
    return c `// conflict detected`
return *No Conflict*;

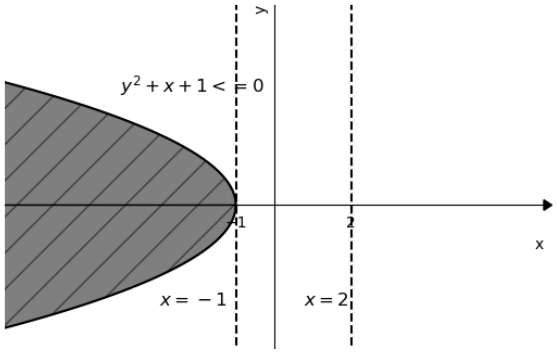

Fig. 1: Demo of a conflict caused by a semantics decision.

one polynomial constraint, making conflicts unavoidable. A demonstration is provided in Example 2.

Example 2. *Consider the formula* $y^2 + x + 1 \leq 0$ *with the variable order* $\{x, y\}$*. As depicted in Figure 2, if we decide* $x \mapsto 2$*, the satisfying region (shaded area) does not intersect the line* $x = 2$*, resulting in a conflict. This conflict is caused by an incorrect semantic decision for variable* x *and could be avoided by choosing a correct value, for instance* $x \mapsto -2$*.*

Algorithm 2: Original NLSAT

Input : A formula F
Output: SAT or UNSAT

while *true* **do**
  $v \leftarrow$ select next variable according to branching heuristic;
  $conf\_cls \leftarrow$ process clauses univariate to v (Algorithm 1);
  if $conf\_cls$ *is empty* **then**
    `// No conflict detected`
    if v *is boolean* **then**
      perform boolean decision;
    else if v *is arithmetic* **then**
      perform semantic decision;
    else
      return *SAT* `// all variables assigned consistently`
  else
    `// Conflict detected`
    $new\_lemma \leftarrow$ conflict analysis via CAD;
    if new_lemma *is empty* **then**
      return *UNSAT* `// formula is unsatisfiable`
    else
      backtrack;

B. Conflicts Caused by Literal Decisions

A key technique in NLSAT is processing clauses that are univariate with respect to the current arithmetic variable. In a CDCL-style systematic search, literals are assigned either through unit propagation for unit clauses (i.e., clauses with only one unassigned literal) or via decisions for clauses with multiple unassigned literals. However, the literal decision mechanism in NLSAT has received relatively little attention. Improper literal decisions may introduce additional conflicts. We illustrate this situation in Example 3.

Example 3. *Consider the following three clauses:*

$$c_1 : y^2 + x - 2 \leq 0 \vee y^2 - x - 2 \leq 0,$$
$$c_2 : x + y = -3, \quad c_3 : x - y = 3.$$

As illustrated in Figure 2, the purple area satisfies both polynomials in c_1*, while the red and blue areas satisfy only* $y^2 + x - 2 \leq 0$ *and* $y^2 - x - 2 \leq 0$*, respectively. The straight lines represent the equality constraints in* c_2 *and* c_3*.*

Suppose the SMT formula is $\{c_1, c_2\}$*. The intersection occurs only in the red area. If the formula is* $\{c_1, c_3\}$*, the intersection is located only in the blue area. When NLSAT processes a clause with multiple unassigned literals such as* c_1*, it decides on one literal and branches the search space into either the red+purple or blue+purple areas. In this scenario,*

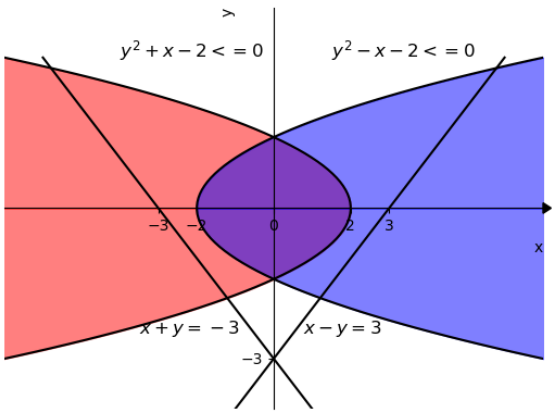

Fig. 2: Demo of a conflict caused by a literal decision.

there is a 50% chance of missing the equality line and encountering a conflict.

However, the feasible-set of c_1 (i.e., the union of red, blue, and purple areas) has a nonempty intersection with both lines, indicating that both formulas $\{c_1, c_2\}$ and $\{c_1, c_3\}$ are indeed satisfiable.

Here comes a new problem in this circumstance. Is it possible to avoid conflicts with a better literal-decision heuristic? If so, how can this be achieved? The key idea is to view the literal-decision problem as a satisfiability problem over intervals. Example 4 illustrates this idea.

Example 4. *Suppose the current assignment is $\alpha := \{x \mapsto 0\}$. Clauses univariate with respect to y are shown as follows:*

$$c_1: \quad (y+2)(y+4) \le x \vee (y-2)(y-4) \le x$$
$$c_2: \quad (y+5)(y+6) \le x \vee (y-1)(y-5) \le x$$

By calculating the feasible-set of each literal, the interval view of the clauses is:

$$c_1 : [-4,-2] \vee [2,4]$$
$$c_2 : [-6,-5] \vee [1,5]$$

Then the problem can be stated as: is there a value that belongs to at least one interval of each clause?

IV. Feasible-Set Based Look-Ahead Mechanism

In this section, we incorporate the clause-level feasible-set into the NLSAT algorithm and design a look-ahead mechanism.

A. Look-Ahead Before Processing Clauses

We first extend the definition of feasible-set to a set of clauses.

Definition 2. *Given a clause set CS, an arithmetic variable x, and an assignment that maps all variables appearing in any clause of CS except x, the* feasible-set *(resp.* infeasible-set*) is the set of values for x that satisfy (resp. unsatisfy) all clauses in CS. Formally, the feasible-set of CS can be computed as the intersection of the feasible-sets of individual clauses:*

$$\text{feasible\_set}(CS) = \bigcap_{c \in CS} \text{feasible\_set}(c)$$

By using the feasible-set of a clause set, we can more directly determine the search space of an arithmetic variable. Specifically, when the feasible-set is non-empty, assigning any value from the set to the variable (i.e., a semantics decision) guarantees progress in the search, allowing the algorithm to proceed to the next stage. Conversely, when the feasible-set is empty, no choice of value can avoid inconsistency.

We now answer the question posed in Section III with the following formal definition:

Definition 3. *Given a clause set, if its feasible-set is non-empty, then, in theory, conflicts can be avoided by choosing appropriate literal assignments; we call this a* path case*. Conversely, if the feasible-set is empty, conflicts cannot be avoided through literal decisions (caused by semantic decisions); we call this a* block case*. Examples 5 and 6 illustrate a path case and a block case, respectively.*

Example 5.

$$\left.\begin{array}{l} c_1 : [-4,-2] \vee \boxed{[2,\ 4]} \to \{[-4,-2] \cup [2,4]\}, \\ c_2 : [-6,-5] \vee \boxed{[1,\ 5]} \to \{[-6,-5] \cup [1,5]\} \end{array}\right\} \bigwedge \to \{[2,4]\}$$

This example shows a path case, since the clauses can be satisfied by deciding literals in the green boxes.

Example 6.

$$\left.\begin{array}{l} c_1 : \boxed{[-4,\ -2]} \vee \boxed{[2,\ 4]} \to \{[-4,-2] \cup [2,4]\}, \\ c_2 : \boxed{[-6,\ -5]} \vee \boxed{[5,\ 6]} \to \{[-6,-5] \cup [5,6]\} \end{array}\right\} \bigwedge \to \varnothing$$

This example shows a block case: the clauses cannot be satisfied regardless of which literals we decide.

The feasible-set computation provides a view of the currently consistent search space. However, in a CDCL-style algorithm, literals still need to be assigned to enable future conflict analysis. This raises the following question: how should we decide literals once we already know it is a path case (i.e., when the green boxes in Example 5 are identified)?

In our approach, we employ a look-ahead mechanism that first selects a pre-appointed value from the feasible-set. This pre-appointed value is then used to guide the search for a consistent decision path. The detailed procedure is presented in Algorithm 3.

The updated algorithm introduces the additional condition that the feasible-set of the current literal must contain the pre-appointed value. This ensures that the feasible-sets of all decided literals during the processing procedure intersect at the clause-set level, allowing the arithmetic variable to be assigned the pre-appointed value. In the block case, the processing algorithm behaves identically to NLSAT, eventually triggering the resolve procedure to revise previous arithmetic assignments.

Algorithm 3: Deciding Literals Using Pre-Appointed Value

Input : A set of clauses F, pre-appointed value val selected from feasible-set
Output: Decided literals $lits$

$lits \leftarrow \emptyset$;
for *each clause* $c \in F$ **do**
  $path\_literal \leftarrow$ undefined;
  for *each literal* $l \in c$ **do**
    $lit\_set \leftarrow$ compute feasible-set of l;
    $val\_lit \leftarrow$ real propagate literal l;
    if $val\_lit = \top$ **then**
      break // clause already satisfied
    if $val\_lit = \bot$ **then**
      continue // literal unsatisfied, check next literal
    if lit_set *contains* val **then**
      $path\_literal \leftarrow l$ // decide satisfiable literal under pre-appointed value
  if *only one literal undefined in* c **then**
    unit propagate the literal;
  else if $path\_literal$ *is defined* **then**
    $lits \leftarrow lits \cup \{path\_literal\}$;
return $lits$;

B. Look-Ahead After Conflict Analysis

In addition to processing clauses with multiple literals, our decision-making algorithm remains effective in conjunction with cylindrical algebraic decomposition (CAD)-based explanation. CAD projects conflict polynomials to learn a lemma that eliminates a sign-invariant cell. The learned lemma includes extended polynomial constraints, referred to as *root atoms*, of the form:

$$y \sim root_i(p(x_1, \ldots, x_n)),$$

where y is the last assigned variable, $\sim \in \{=, \neq, \leq, \geq, <, >\}$, and p is a polynomial generated via model-based projection, involving previously assigned variables $x_1, \ldots, x_n$. Specifically, when the last assigned variable y lies between two polynomial constraints, multiple root atoms may be generated in the learned lemma, making the choice of literal assignment particularly critical.

Example 7.

c_1 : [-7, -2] $\vee$ $[2, 8]$ $\quad$ $c_1 : [-7, -2] \vee$ [2, 8]
c_2 : $[-11, -10] \vee$ [-6, 5] $\quad$ $c_2 : [-11, -10] \vee$ [-6, 5]
$learned$: [3, 4] $\vee$ [7, 8] $\quad$ $learned :$ [3, 4] $\vee [7, 8]$

Before learning a new lemma, the left column highlights a possible path in the green boxes. After incorporating the new lemma, the updated feasible-set may become inconsistent with any literal in the lemma, resulting in a conflict. The right column shows a new feasible path after incrementally processing the lemma.

Algorithm 4: Process Clauses After a New Lemma

Input: A new lemma $lemma$, arithmetic variable v

$lemma\_feasible\_set \leftarrow$ compute feasible-set of clause($lemma$);
$feasible\_set[v] \leftarrow feasible\_set[v] \cap lemma\_feasible\_set$;
if $feasible\_set[v]$ *is empty* **then**
  // Block case: no consistent assignment possible for v
  // Proceed as in original NLSAT (Algorithm 1)
  original process clauses;
else
  // Path case: a consistent assignment exists for v
  $val \leftarrow$ value_selection($feasible\_set[v]$);
  // Call Algorithm 3 to decide literals using the pre-appointed value
  deciding literals using pre-appointed value (val);

Look-ahead algorithm after conflict analysis is similar to the main search part, as shown in Algorithm 4. The main difference is the incremental computation of decision cases by considering only the feasible-set of the new lemma. We use a vector of intervals to cache the previous feasible sets. After calculating the clause-level feasible-set, we must reprocess the clauses for the path case and find a new decision path. Because the newly generated lemma adds a new constraint on the arithmetic variable, the current decision path might be blocked as shown in Example 7.

V. Clause-Level Propagation

Following the idea of using a clause-level feasible-set, this section introduces a new kind of propagation called *clause-level propagation*. The idea is inspired by unit propagation (or literal propagation) in SAT solving. In a boolean satisfaction problem, boolean variables can be unit propagated to assign a value, which allows the solver to detect conflicts as early as possible. However, most existing complete algorithms do not perform arithmetic propagation for quick assignment or conflict detection.

We now formally define clause-level propagation.

Definition 4. *Given a clause c, an arithmetic variable x, and an assignment α that assigns all variables appearing in c except x,* clause-level propagation *on x is the computation of the feasible-set of the clause c, which serves to narrow the feasible-set of the variable x.*

The main difference between arithmetic and boolean problems lies in the structure of their search spaces. In SAT solving, unit propagation assigns boolean variables to either true or false. In other words, unit propagation always prunes the search space of a boolean variable by half, effectively guiding the search forward, since there are only two possible values.

By contrast, in arithmetic problems, a clause may only eliminate part of the real-valued domain of a variable, contracting the search space without fully deciding the variable. In this section, we introduce the *clause-level propagation* algorithm, which computes feasible-sets for arithmetic variables, and then show how to use this propagation information to guide the search by selecting the next branching variable.

A. Clause-Level Propagation Method

In SAT solving, unit propagation is performed after variable assignments. In our algorithm, we incrementally compute the feasible-set of a clause whenever it becomes univariate with respect to an arithmetic variable[5].

A newly generated univariate clause imposes an additional constraint on the arithmetic variable, pruning its search space by taking the intersection with the existing feasible-set. Unlike boolean search spaces, arithmetic search spaces may only be partially reduced. We categorize clause-level propagation into three cases, illustrated in Example 8:

- **Block case:** The clause-level feasible-set is empty.
- **Fixed case:** The clause-level feasible-set contains exactly one real number, e.g., $[2, 2]$.
- **Other case:** The clause-level feasible-set is narrowed but neither empty nor a single value.

This propagation information is subsequently used to guide the branching heuristic. The algorithmic details are shown in Algorithm 5.

Example 8. *An example is shown in Figure 3. When a variable x is assigned 0, three clauses become univariate to other variables $\{z, y, k\}$. These three clauses add three new constraints on arithmetic variables, calculated as*

$$\{(-\infty, -2] \cup [2, 6]\}, \quad \{[2, 2]\}, \quad \emptyset.$$

*These feasible sets indicate that variables are **feasible**, **fixed**, or **blocked**.*

B. Propagation-Based Branching Heuristic

In SAT solving, after unit propagation, an unassigned boolean variable is either propagated to a value or a conflict clause is detected immediately. Similarly, for arithmetic variables, the clause-level feasible-set allows us to identify propagation and conflict cases, corresponding to the *fixed* and *blocked* cases introduced above.

However, unlike boolean unit propagation, clause-level conflicts for arithmetic variables cannot directly return a conflict clause in NLSAT; this task remains the responsibility of the clause processing procedure. Therefore, we record information about fixed and blocked variables and prioritize them in the branching heuristic, ensuring that the search addresses these critical variables as early as possible.

For variables in the normal case (neither fixed nor blocked), we adopt the Variable State Independent Decaying Sum (VSIDS) heuristic [27], as suggested in [23], [28]. The full procedure is outlined in Algorithm 6. Details on our implementation of dynamic variable ordering are discussed in Section VI.

[5]This occurs not only after assigning an arithmetic variable, but also after assigning a boolean variable. Whenever a clause is arithmetically univariate, its feasible-set is updated.

Algorithm 5: Clause-Level Propagation

Input: Clause set F, current feasible sets of arithmetic variables

```
for each clause cls ∈ F do
    if cls is univariate to an arithmetic variable v then
        cls_feasible_set ← compute_feasible_set(cls);
        feasible_set[v] ←
          feasible_set[v] ∩ cls_feasible_set;
        // Categorize propagation result
        if feasible_set[v] is empty then
            blocked_vars ← blocked_vars ∪ {v} ;
             // Block case: conflict
             unavoidable
        else if feasible_set[v] is a single value then
            fixed_vars ← fixed_vars ∪ {v} ;
             // Fixed case: value
             determined
        else
            // Other case: search space
               narrowed but not fixed
```

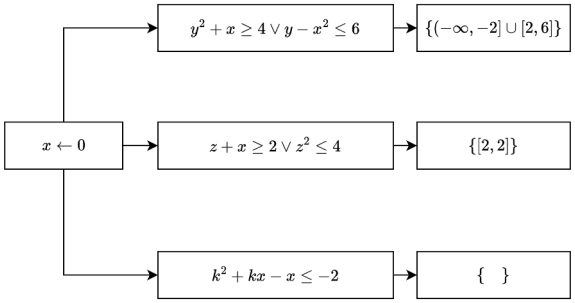

Fig. 3: Demo of clause-level propagation for y (**normal case**), z (**fixed case**) and k (**block case**).

VI. Implementation

All of the above algorithms are incorporated into our new solver, `clauseSMT`. We describe the solver in detail, including an extended version of conflict analysis and an implementation of the dynamic variable ordering framework. The overall structure of `clauseSMT` is illustrated in Figure 4, and

Algorithm 6: Propagation-Based Branching Heuristic

Output: A variable v to branch on

```
// Prioritize blocked variables (potential conflicts) first
if blocked_vars ≠ ∅ then
    v ← select_from(blocked_vars);
// Next, consider fixed variables (propagate value)
else if fixed_vars ≠ ∅ then
    v ← select_from(fixed_vars);
// Otherwise, use VSIDS heuristic for normal variables
else
    v ← vsids_select();
return v
```

Algorithm 7: ClauseSMT

Input : A formula F

Output: SAT or UNSAT

```
while true do
    clause-level propagation(F);
    // call Algorithm 5
    variable v ← branching heuristic;
    // call Algorithm 6
    if v's feasible-set is empty then
        new_lemma ← Resolve (Conflict Analysis);
        if new_lemma is empty then
            return UNSAT;
        else
            Process Clauses after a new lemma(new_lemma);
            // call Algorithm 4
    else
        val ← select from feasible-set;
        Process Clauses using pre-appointed value val;
        // call Algorithm 3
        assign v ← val;
        if all variables are assigned then
            return SAT;
```

Algorithm 7 presents the integrated heuristics implemented in the solver.

A. Resolve

Although the look-ahead based processing algorithm is introduced in the previous section, practical implementation involves more complex cases. As discussed, conflicts in NLSAT can be categorized into two types:

1) **Literal-decision conflicts** [6]

[6]For the look-ahead mechanism, this occurs only in block cases.

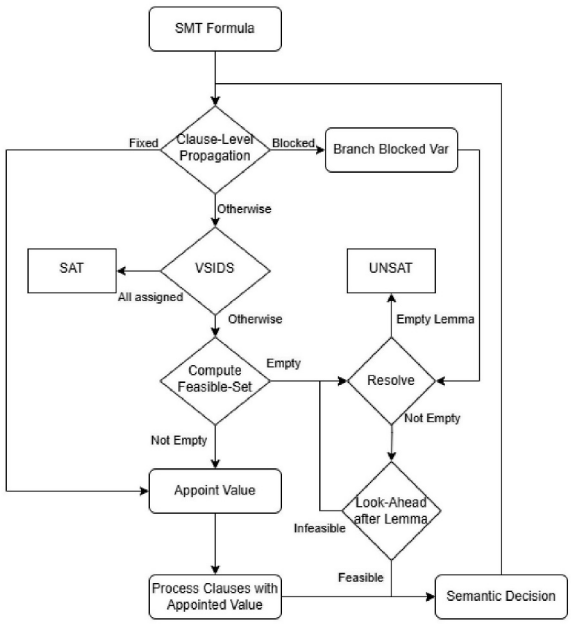

Fig. 4: Overall Structure of clauseSMT.

For these, we keep the resolve algorithm the same as in the original NLSAT. The search engine backtracks to the highest decision level of the new lemma and attempts to unit propagate the negation of previously decided literals. In other words, this type of conflict occurs purely at the literal level.

2) **Incremental clause conflicts**
For literal-decision conflicts considering an incremental clause (i.e., a newly learned lemma), we detect these by recalculating the decision path. Any literals that were decided prior to this stage must be reset to allow proper propagation according to the updated feasible sets.

To better manage the backtracking process, we introduce several new types of trails, commonly used in the NLSAT algorithm:

- **path_finder**: Whenever the feasible-set at the current stage is non-empty, a path exists and this trail is recorded.
- **block_finder**: When the current stage is blocked, this trail is recorded.
- **clause_feasible_updated**: Whenever the feasible-set of a clause for an arithmetic variable is updated, this trail is recorded.

B. Dynamic Variable Ordering Framework

1) Watched Variables: We implement two *watched variables*, inspired by the two-watched-literals scheme, to detect univariate clauses and lemmas. Each clause or lemma is watched by two variables (boolean or arithmetic) appearing in it. Watchers are updated whenever one of them is assigned. The cases are as follows:

- There exists a third variable unassigned: we replace the assigned watcher with this variable.
- No other variable is unassigned: this clause is univariate to the remaining unassigned watcher.
- Both watchers are assigned: no action is taken.

Whenever a univariate clause is detected, its feasible-set is updated eagerly, which helps the search engine gather more clause-level information.

2) Projection Order: A common challenge in nonlinear arithmetic is the strict variable-order relationship of root atoms, which are generated by model-based projection. Given a polynomial set ps and a projection order $\{v_1, v_2, \ldots, v_k\}$, each time the projection method eliminates a variable, it generates a root atom corresponding to that variable.

As discussed in [23], in most cases, the variable not appearing in the polynomial should be assigned last. Specifically, when all atoms are in root format, the projection order should exactly be the inverse of the assignment order, which is how it is implemented in our solver.

3) Branching Heuristic: VSIDS is a particularly effective branching heuristic. Each time a conflict is detected, we increase the activities of the involved variables of all types. Following the design in [23], we employ several branching heuristics and use the **uniform** heuristic to compare the activity of boolean and arithmetic variables on the same scale.

4) Parameter Settings: Values of the tunable parameters are summarized in Table I.

Symbol	Description	Value
$arith\_decay$	Decay factor for arithmetic variables	0.95
$bool\_decay$	Decay factor for boolean variables	0.95
$arith\_bump$	Incremental amount of arithmetic activity	1
$bool\_bump$	Incremental amount of boolean activity	1
$lemma\_conf$	Initial conflict count for deleting lemmas	100
$lemma\_conf\_inc$	Incremental factor for lemma conflicts	1.5

TABLE I: Tunable parameters

C. Shortcut for UNSAT Instances

For the block case discussed in Section IV, we process blocked clauses the same way as in NLSAT. As concluded earlier, conflicts in this scenario occur because previous variables (arithmetic or boolean) were assigned incorrect values. In our implementation, we introduce a shortcut mechanism to directly return `UNSAT` if the blocked clauses involve only a single variable. In this situation, there are no previous stages, and thus the instance is guaranteed to be unsatisfiable.

VII. Evaluation

In this section, we compare our algorithm with several existing solvers, including Z3 (version 4.13.1) [25], CVC5 (version 1.0.2) [29], and YICES2 (version 2.6.2) [30]. We also present an ablation study to analyze the impact of various improvements.

A. Experiment Preliminaries

The standard benchmark for evaluating SMT solvers is SMT-LIB[7]. The full benchmark for the QF_NRA theory consists of 12,134 instances, originating from various applications, including nonlinear hybrid automata, ranking function generation for program analysis, and other mathematical problems. Most instances are labeled as `SAT` or `UNSAT`, though some remain `UNKNOWN`. It should be noted that instances from SMT-LIB exhibit significant variation in clause numbers, literal counts, and polynomial degrees. Our experiments are conducted on a server equipped with an Intel Xeon Platinum 8153 processor running at 2.00 GHz. Each instance is limited to a maximum runtime of 1,200 seconds, consistent with the SMT-COMP settings.

B. RQ1: Comparison with mainstream Solvers

We compare our algorithm with other SMT solvers in Table II. When evaluating our approach using Z3, we disable all other tactics such as CDCL(T) and any incomplete algorithms. The solvers Z3, CVC5, and YICES2 are tested without modifications, each employing its portfolio of different algorithms. We also evaluate the original NLSAT solver by disabling all other tactics.

Our algorithm demonstrates competitive performance compared to state-of-the-art solvers such as Z3 and CVC5. In particular, clauseSMT solves the largest number of satisfiable instances and ranks third for unsatisfiable ones.

Figure 5 presents pairwise scatter plots of solving times, where each point's x-coordinate corresponds to clauseSMT and the y-coordinate to a competing solver. Points below the diagonal ($y = x$) indicate instances where clauseSMT is faster.

The plots show that clauseSMT efficiently handles satisfiable instances (blue points), often outperforming mainstream solvers, consistent with the results in Table II. The comparison with the original NLSAT baseline further highlights the benefits of our look-ahead and arithmetic-aware propagation techniques, which reduce solving time on most instances.

1) Comparison with CVC5: CVC5 solves the largest number of instances overall, particularly excelling in unsatisfiable cases due to incomplete techniques such as interval constraint propagation and incremental linearization. Notably, the MBO category [31], which contains single clauses with very high degrees, remains challenging for CAD-based algorithms.

2) Comparison with Original NLSAT: Our solver performs slightly worse on instances with high-degree polynomials, where feasible-set computations are relatively costly. Nevertheless, it demonstrates significant gains in LassoRanker and Hycomp, which contain thousands of instances with complex feasible-set relationships. For LassoRanker, our solver improves solved instances by 50%, and overall it solves almost 300 more instances compared to the original NLSAT.

[7]https://smt-lib.org/

Category	#inst		Z3	YICES2	CVC5	NLSAT	Ours
20161105-Sturm-MBO	405	SAT	0	0	0	0	0
		UNSAT	124	**285**	**285**	44	39
		SOLVED	124	**285**	**285**	44	39
20161105-Sturm-MGC	9	SAT	**2**	0	0	**2**	**2**
		UNSAT	**7**	0	0	**7**	6
		SOLVED	**9**	0	0	**9**	8
20170501-Heizmann	69	SAT	**2**	0	1	1	**2**
		UNSAT	1	12	9	10	**19**
		SOLVED	3	12	10	11	**21**
20180501-Economics-Mulligan	135	SAT	**93**	91	89	**93**	92
		UNSAT	39	39	35	**41**	**41**
		SOLVED	132	130	124	**134**	131
2019-ezsmt	63	SAT	56	52	50	**58**	36
		UNSAT	**2**	**2**	**2**	**2**	**2**
		SOLVED	58	54	52	**60**	38
20200911-Pine	245	SAT	234	**235**	199	**235**	234
		UNSAT	6	**8**	5	7	5
		SOLVED	240	**243**	204	242	239
20211101-Geogebra	112	SAT	**110**	99	91	**110**	98
		UNSAT	0	0	0	0	0
		SOLVED	**110**	99	91	**110**	98
20220314-Uncu	225	SAT	69	**70**	62	68	**70**
		UNSAT	**155**	153	148	**155**	152
		SOLVED	**224**	223	210	223	222
hong	20	SAT	0	0	0	0	0
		UNSAT	8	**20**	**20**	12	14
		SOLVED	8	**20**	**20**	12	14
hycomp	2752	SAT	**307**	227	225	244	291
		UNSAT	**2242**	2201	2212	2088	2181
		SOLVED	**2549**	2428	2437	2332	2472
kissing	45	SAT	**33**	10	17	12	14
		UNSAT	0	0	0	0	0
		SOLVED	**33**	10	17	12	14
LassoRanker	821	SAT	167	122	**305**	220	302
		UNSAT	151	260	**470**	174	311
		SOLVED	318	382	**775**	394	613
meti-tarski	7006	SAT	**4391**	4369	4343	**4391**	4372
		UNSAT	2605	2588	2581	**2611**	2588
		SOLVED	6996	6957	6924	**7002**	6960
UltimateAutomizer	61	SAT	35	39	35	**45**	39
		UNSAT	11	12	10	**13**	12
		SOLVED	46	51	45	**58**	51
zankl	166	SAT	**70**	58	58	62	56
		UNSAT	28	**32**	**32**	27	30
		SOLVED	**98**	90	90	89	86
Total	12134	SAT	5569	5372	5475	5541	**5608**
		UNSAT	5379	5612	**5809**	5191	5397
		SOLVED	10948	10984	**11284**	10732	11005

TABLE II: Summary of results for all instances in SMT-LIB (QF_NRA).

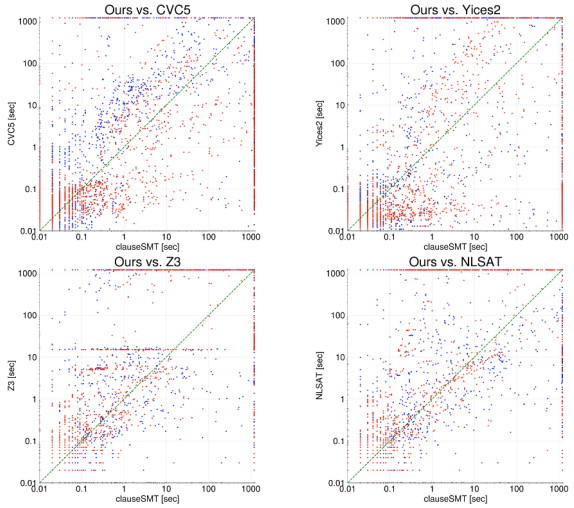

Fig. 5: Run time comparison against CVC5, Z3, YICES2 and nlsat (blue points: satisfiable instances, red points: unsatisfiable instances).

C. RQ2: Effectiveness of Look-Ahead Mechanism

To evaluate the impact of the look-ahead mechanism, we implement several variants summarized in Table III:

- **Look-Ahead**: Feasible-set based look-ahead on original NLSAT with static variable order.
- **Lower Degree**: Decide literals with the lowest polynomial degree (default NLSAT heuristic).
- **Random Decide**: Randomly select literals when processing clauses.

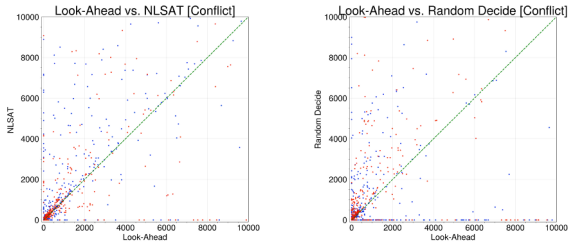

Fig. 6: Conflict counts of look-ahead NLSAT versus original and random NLSAT (blue: satisfiable, red: unsatisfiable).

Category	#inst	Decide Lower Degree	Random Decide	Look-Ahead
20161105-Sturm-MBO	405	44	45	44
20161105-Sturm-MGC	9	9	9	9
20170501-Heizmann	69	11	5	7
20180501-Economics-Mulligan	135	134	134	134
2019-ezsmt	63	60	59	58
20200911-Pine	245	242	242	243
20211101-Geogebra	112	110	109	110
20220314-Uncu	225	223	224	224
hong	20	12	12	12
hycomp	2752	2332	2272	2388
kissing	45	12	14	15
LassoRanker	821	394	393	389
meti-tarski	7006	7002	7001	7002
UltimateAutomizer	61	58	44	57
zankl	166	89	89	87
Total	12134	10732	10652	10778

TABLE III: Comparison of solved instances for different literal decision mechanisms.

Although the difference in solved instances across most categories is small, our algorithm solves about 50 more instances in the Hycomp category [18], which contains numerous nonlinear equalities. These instances often exhibit literal path cases, highlighting the advantage of the look-ahead mechanism.

The look-ahead mechanism proactively detects blocking cases and mitigates conflicts during path exploration. Figure 6 presents a scatter plot comparing the number of conflicts incurred by the two algorithms, with the green line indicating parity. For most instances, the look-ahead strategy reduces conflicts, which—although not substantially impacting the 1200-second timeout—enables a more efficient systematic search, particularly when handling clauses with multiple literals. Interestingly, a few instances exhibit increased conflicts under the look-ahead mechanism. This arises from differences in sampling intervals used to select arithmetic assignments: the look-ahead algorithm employs the intersected interval for witness selection, whereas the baseline relies on the literal interval. Consequently, even with identical random seeds, the arithmetic variables may be assigned different values, leading to divergent subsequent search processes.

D. RQ3: Effectiveness of Clause-Level Propagation

To evaluate clause-level propagation, we compare three solver versions: (1) original NLSAT with static variable order

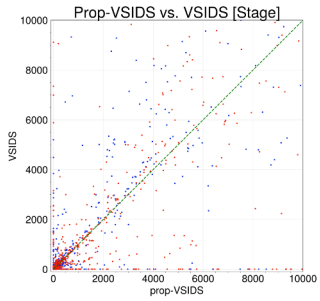

Fig. 7: Stage comparison of prop-VSIDS against VSIDS (blue points: satisfiable instances, red points: unsatisfiable instances).

Category	#inst	Static	VSIDS	prop-VSIDS
20161105-Sturm-MBO	405	44	38	39
20161105-Sturm-MGC	9	9	6	8
20170501-Heizmann	69	7	20	21
20180501-Economics-Mulligan	135	134	133	133
2019-ezsmt	63	58	30	38
20200911-Pine	245	243	239	239
20211101-Geogebra	112	110	101	98
20220314-Uncu	225	224	222	222
hong	20	12	11	11
hycomp	2752	2388	2426	2472
kissing	45	15	14	14
LassoRanker	821	389	571	613
meti-tarski	7006	7002	6974	6960
UltimateAutomizer	61	57	52	51
zankl	166	87	83	86
Total	12134	10778	10920	11005

TABLE IV: Comparison of solved instances for different branching heuristics.

based on degree (*static*), (2) dynamic NLSAT with VSIDS (*VSIDS*), and (3) dynamic NLSAT with clause-level propagation (*prop-VSIDS*). Results are summarized in Table IV.

The data show that VSIDS significantly improves performance within the MCSAT framework. Incorporating clause-level propagation further accelerates conflict detection and increases the number of solved instances across most categories. In particular, *prop-VSIDS* excels on `hycomp` [18], `LassoRanker` [16], [17], and `meti-tarski` [32], all of which contain numerous arithmetic clauses prone to block cases. Figure 7 illustrates the reduction in search stages (semantic decision steps) for *prop-VSIDS* compared to traditional VSIDS, showing that inconsistent branching choices are detected earlier and overall stages are significantly reduced.

E. Threats to Validity

Correctness of implementation. Developing `clauseSMT` required substantial effort. All comparisons with other solvers were executed in the same environment. Results were carefully verified, and satisfiable instances were validated to ensure correctness.

Randomness. NLSAT uses internal randomness for semantic decisions and clause/literal reordering. These mechanisms are preserved in our solver, so they do not affect comparisons. Additionally, a fully random literal-decision variant was tested (and found uncompetitive, as discussed). Key metrics such as conflict counts and search stages are reported via scatter plots.

VIII. Related Work

SMT-solving methods can be categorized into complete and incomplete approaches. Incomplete methods are fast due to specialized techniques. Interval constraint propagation (ICP) [33], [34], as implemented in dReal [35], is widely used for quickly detecting unsatisfiable instances. Local search has also been extended from SAT to arithmetic theories, including integer [36], [37], linear/multilinear real [22], and nonlinear real arithmetic [26], [38].

Complete methods dominate modern SMT solvers, performing well on both satisfiable and unsatisfiable instances. CDCL(T) [39] and NLSAT [21] rely on CAD for theory reasoning [40], while MCSAT maintains high performance across diverse applications using lighter explanation modules [41]. Recent work has focused on improving NLSAT efficiency, e.g., by optimizing variable projection orders [42]–[44], designing innovative projection operators [45], generating larger literal-invariant cells [46], [47], and exploring dynamic branching heuristics [23].

IX. Conclusion

We presented a clause-level NLSAT algorithm for SMT solving over nonlinear real arithmetic. We categorized conflicts in NLSAT and analyzed the challenges of literal decisions faced by many CDCL-style algorithms. To address these challenges, we introduced a feasible-set-based look-ahead mechanism and clause-level propagation for branching. Experimental results show that our solver is competitive with mainstream solvers, demonstrating the effectiveness of the proposed techniques.

In future work, we aim to develop a clause-level approach for block cases, which are closely related to quantifier elimination. We anticipate that lighter alternatives to CAD may allow connecting literals across clauses, revising previous decisions, and constructing consistent decision paths.

X. Acknowledgments

The author gratefully acknowledges the use of computing resources provided by the Institute of Software, Chinese Academy of Sciences. The author also sincerely thanks the anonymous reviewers for their constructive comments and suggestions, which helped to improve the quality of this paper.

References

[1] C. W. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Springer, 2018, pp. 305–343. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_11

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[4] P. Yao, Q. Shi, H. Huang, and C. Zhang, "Program analysis via efficient symbolic abstraction," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485495

[5] D. Beyer, M. Dangl, and P. Wendler, "A unifying view on smt-based software verification," *J. Autom. Reason.*, vol. 60, no. 3, p. 299–335, mar 2018. [Online]. Available: https://doi.org/10.1007/s10817-017-9432-6

[6] J. Wang and C. Wang, "Learning to synthesize relational invariants," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556942

[7] M. Tappler, B. K. Aichernig, and F. Lorber, "Timed automata learning via smt solving," in *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 489–507. [Online]. Available: https://doi.org/10.1007/978-3-031-06773-0_26

[8] R. Xu, J. An, and B. Zhan, "Active learning of one-clock timed automata using constraint solving," in *Automated Technology for Verification and Analysis: 20th International Symposium, ATVA 2022, Virtual Event, October 25–28, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 249–265. [Online]. Available: https://doi.org/10.1007/978-3-031-19992-9_16

[9] G. Amir, H. Wu, C. Barrett, and G. Katz, "An smt-based approach for verifying binarized neural networks," in *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 – April 1, 2021, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 203–222. [Online]. Available: https://doi.org/10.1007/978-3-030-72013-1_11

[10] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 97–117.

[11] B. Paulsen and C. Wang, "Linsyn: Synthesizing tight linear bounds for arbitrary neural network activation functions," in *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 357–376. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_19

[12] ——, "Example guided synthesis of linear approximations for neural network verification," in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 149–170. [Online]. Available: https://doi.org/10.1007/978-3-031-13185-1_8

[13] K. Bae and S. Gao, "Modular smt-based analysis of nonlinear hybrid systems," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, 2017, pp. 180–187.

[14] Y. Shoukry, M. Chong, M. Wakaiki, P. Nuzzo, A. Sangiovanni-Vincentelli, S. A. Seshia, J. A. P. Hespanha, and P. Tabuada, "Smt-based observer design for cyber-physical systems under sensor attacks," *ACM Trans. Cyber-Phys. Syst.*, vol. 2, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3078621

[15] A. Cimatti, "Application of smt solvers to hybrid system verification," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 4–4.

[16] J. Leike and M. Heizmann, "Ranking templates for linear loops," *Log. Methods Comput. Sci.*, vol. 11, no. 1, 2015. [Online]. Available: https://doi.org/10.2168/LMCS-11(1:16)2015

[17] M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski, "Linear ranking for linear lasso programs," in *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, ser. Lecture Notes in Computer Science, D. V. Hung and M. Ogawa, Eds., vol. 8172. Springer, 2013, pp. 365–380. [Online]. Available: https://doi.org/10.1007/978-3-319-02444-8_26

[18] A. Cimatti, S. Mover, and S. Tonetta, "A quantifier-free SMT encoding of non-linear hybrid automata," in *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, G. Cabodi and S. Singh, Eds. IEEE, 2012, pp. 187–195. [Online]. Available: https://ieeexplore.ieee.org/document/6462573/

[19] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[20] B. F. Caviness and J. R. Johnson, "Quantifier elimination and cylindrical algebraic decomposition," in *Texts and Monographs in Symbolic Computation*, 2004.

[21] D. Jovanovic and L. M. de Moura, "Solving non-linear arithmetic," in *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, ser. Lecture Notes in Computer Science, B. Gramlich, D. Miller, and U. Sattler, Eds., vol. 7364. Springer, 2012, pp. 339–354. [Online]. Available: https://doi.org/10.1007/978-3-642-31365-3_27

[22] B. Li and S. Cai, "Local search for smt on linear and multi-linear real arithmetic," in *2023 Formal Methods in Computer-Aided Design (FMCAD)*, 2023, pp. 1–10.

[23] J. Nalbach, G. Kremer, and E. Ábrahám, "On variable orderings in mcsat for non-linear real arithmetic," in *SC-square@SIAM AG*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204767299

[24] F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám, "Smt-rat: An open source c++ toolbox for strategic and parallel smt solving," in *Theory and Applications of Satisfiability Testing – SAT 2015*, M. Heule and S. Weaver, Eds. Cham: Springer International Publishing, 2015, pp. 360–368.

[25] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[26] H. Li, B. Xia, and T. Zhao, "Local search for solving satisfiability of polynomial formulas," in *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, ser. Lecture Notes in Computer Science, C. Enea and A. Lal, Eds., vol. 13965. Springer, 2023, pp. 87–109. [Online]. Available: https://doi.org/10.1007/978-3-031-37703-7_5

[27] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[28] D. Jovanovic, C. Barrett, and L. de Moura, "The design and implementation of the model constructing satisfiability calculus," in *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 173–180.

[29] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: https://doi.org/10.1007/978-3-030-99524-9_24

[30] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_49

[31] T. Akutsu, M. Hayashida, and T. Tamura, "Algorithms for inference, analysis and control of boolean networks," in *Algebraic Biology, Third International Conference, AB 2008, Castle of Hagenberg, Austria, July 31-August 2, 2008, Proceedings*, ser. Lecture Notes in Computer Science, K. Horimoto, G. Regensburger, M. Rosenkranz, and H. Yoshida, Eds., vol. 5147. Springer, 2008, pp. 1–15. [Online]. Available: https://doi.org/10.1007/978-3-540-85101-1_1

[32] B. Akbarpour and L. C. Paulson, "Metitarski: An automatic theorem prover for real-valued special functions," *J. Autom. Reason.*,

vol. 44, no. 3, pp. 175–205, 2010. [Online]. Available: https://doi.org/10.1007/s10817-009-9149-2

[33] T. V. Khanh and M. Ogawa, "SMT for polynomial constraints on real numbers," in *Third Workshop on Tools for Automatic Program Analysis, TAPAS 2012, Deauville, France, September 14, 2012*, ser. Electronic Notes in Theoretical Computer Science, B. Jeannet, Ed., vol. 289. Elsevier, 2012, pp. 27–40. [Online]. Available: https://doi.org/10.1016/j.entcs.2012.11.004

[34] V. X. Tung, T. V. Khanh, and M. Ogawa, "raSAT: an SMT solver for polynomial constraints," *Formal Methods Syst. Des.*, vol. 51, no. 3, pp. 462–499, 2017. [Online]. Available: https://doi.org/10.1007/s10703-017-0284-9

[35] S. Gao, S. Kong, and E. M. Clarke, "dreal: An SMT solver for nonlinear theories over the reals," in *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. P. Bonacina, Ed., vol. 7898. Springer, 2013, pp. 208–214. [Online]. Available: https://doi.org/10.1007/978-3-642-38574-2_14

[36] S. Cai, B. Li, and X. Zhang, "Local search for SMT on linear integer arithmetic," in *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part II*, ser. Lecture Notes in Computer Science, S. Shoham and Y. Vizel, Eds., vol. 13372. Springer, 2022, pp. 227–248. [Online]. Available: https://doi.org/10.1007/978-3-031-13188-2_12

[37] ——, "Local search for satisfiability modulo integer arithmetic theories," *ACM Trans. Comput. Logic*, vol. 24, no. 4, jul 2023. [Online]. Available: https://doi.org/10.1145/3597495

[38] Z. Wang, B. Zhan, B. Li, and S. Cai, "Efficient local search for nonlinear real arithmetic," in *Verification, Model Checking, and Abstract Interpretation: 25th International Conference, VMCAI 2024, London, United Kingdom, January 15–16, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 326–349. [Online]. Available: https://doi.org/10.1007/978-3-031-50524-9_15

[39] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *J. ACM*, vol. 53, no. 6, pp. 937–977, 2006. [Online]. Available: https://doi.org/10.1145/1217856.1217859

[40] G. Kremer, "Cylindrical algebraic decomposition for nonlinear arithmetic problems," Ph.D. dissertation, RWTH Aachen University, Germany, 2020. [Online]. Available: https://publications.rwth-aachen.de/record/792185

[41] L. M. de Moura and D. Jovanovic, "A model-constructing satisfiability calculus," in *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds., vol. 7737. Springer, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1007/978-3-642-35873-9_1

[42] H. Li, B. Xia, H. Zhang, and T. Zheng, "Choosing better variable orderings for cylindrical algebraic decomposition via exploiting chordal structure," *J. Symb. Comput.*, vol. 116, pp. 324–344, 2023. [Online]. Available: https://doi.org/10.1016/j.jsc.2022.10.009

[43] C. Chen, Z. Zhu, and H. Chi, "Variable ordering selection for cylindrical algebraic decomposition with artificial neural networks," in *Mathematical Software – ICMS 2020: 7th International Conference, Braunschweig, Germany, July 13–16, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 281–291. [Online]. Available: https://doi.org/10.1007/978-3-030-52200-1_28

[44] F. Jia, Y. Dong, M. Liu, P. Huang, F. Ma, and J. Zhang, "Suggesting variable order for cylindrical algebraic decomposition via reinforcement learning," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=vNsdFwjPtL

[45] H. Li and B. Xia, "Solving satisfiability of polynomial formulas by sample-cell projection," 2020.

[46] E. Ábrahám, J. H. Davenport, M. England, and G. Kremer, "Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings," *J. Log. Algebraic Methods Program.*, vol. 119, p. 100633, 2021. [Online]. Available: https://doi.org/10.1016/j.jlamp.2020.100633

[47] J. Nalbach, E. Ábrahám, P. Specht, C. W. Brown, J. H. Davenport, and M. England, "Levelwise construction of a single cylindrical algebraic cell," *J. Symb. Comput.*, vol. 123, no. C, may 2024. [Online]. Available: https://doi.org/10.1016/j.jsc.2023.102288