Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems

Sicun Gao[1,2], Malay Ganai[1], Franjo Ivančić[1], Aarti Gupta[1], Sriram Sankaranarayanan[3], and Edmund M. Clarke[2]
1 NEC Labs America, NJ, USA 2 Carnegie Mellon University, PA, USA 3 University of Colorado Boulder, CO, USA

***Abstract*—We propose a novel integration of interval constraint propagation (ICP) with SMT solvers for linear real arithmetic (LRA) to decide nonlinear real arithmetic problems. We use ICP to search for interval solutions of the nonlinear constraints, and use the LRA solver to either validate the solutions or provide constraints to incrementally refine the search space for ICP. This serves the goal of separating the linear and nonlinear solving stages, and we show that the proposed methods preserve the correctness guarantees of ICP. Experimental results show that such separation is useful for enhancing efficiency.**

I. Introduction

Formal verification of embedded software and hybrid systems often requires deciding satisfiability of quantifier-free first-order formulas involving real number arithmetic. While highly efficient algorithms [10] exist for deciding linear real arithmetic (QFLRA problems, as named in SMT-LIB [5]), nonlinear formulas (QFNRA problems [5]) have been a major obstacle in the scalable verification of realistic systems. Existing complete algorithms have very high complexity for nonlinear formulas with polynomial functions (double-exponential lower bound [9]). Formulas containing transcendental functions are in general undecidable. It is thus important to find alternative practical solving techniques for which the completeness requirement may be relaxed to some extent ([11], [12], [20], [8]).

Interval Constraint Propagation (ICP) is an efficient numerical method for finding interval over-approximations of solution sets of nonlinear real equality and inequality systems ([15], [6]). For solving QFNRA formulas in a DPLL(T) framework, ICP can be used as the theory solver that provides decisions on conjunctions of theory atoms. What distinguishes ICP from other numerical solution-finding algorithms (such as Newton-Raphson or convex optimization) is that it guarantees the following reliability properties:

- ICP always terminates, returning either "unsatisfiable", or "satisfiable" with an interval overapproximation of a solution (or the solution set).
- When ICP returns an "unsatisfiable" decision, it is always correct.
- When ICP returns a "satisfiable" decision, the solution may be spurious; but its error is always within a given bound that can be set very small.

A detailed discussion of what these correctness guarantees of ICP imply for decision problems is given in Section IV. These properties ensure that an ICP solver only relaxes completeness *moderately* (see "δ-completeness", Section IV), while achieving efficiency. ICP algorithms have been applied to various nonlinear scientific computing problems involving thousands of variables and constraints (including transcendental functions) ([17], [18], [6]).

The HySAT/iSAT solver [11] is a state-of-the-art SMT solver for QFNRA problems. HySAT uses ICP for handling nonlinear real constraints. It carefully builds Boolean solving capacities into ICP by exploiting the similarity between SAT and interval constraint solving algorithms. HySAT successfully solved many challenging nonlinear benchmarks that arise from hybrid system verification problems [3].

However, a problem with HySAT is that it handles both linear and nonlinear constraints with ICP. It is known that ICP does not solve linear constraints efficiently enough. In fact, ICP can suffer from the "slow convergence" problem [7] on easy linear constraints such as "$x \geq y \wedge x \leq y$", where it needs a large number of iteration steps to return an answer. As there exist highly optimized algorithms for deciding linear arithmetic problems [10], solving all the constraints in ICP is suboptimal. Most practical formal verification problems contain a large number of linear and Boolean constraints, and only a small number of nonlinear ones. Ideally, we would like to solve linear and nonlinear constraints differently, and apply the efficient algorithms for linear constraints as much as possible.

Such separation of linear and nonlinear solving is not straightforward to design. In fact, it is suggested as an open question in the original HySAT paper [11]. There are several difficulties involved:

- The linear and nonlinear constraints share many variables in nontrivial problems. For the same variable, the linear solver returns point solutions while ICP returns interval solutions. It is not straightforward to check consistency between the different solutions.
- As both the linear solver and the nonlinear solver return only one solution (point or interval box) at a time, it is impossible to enumerate all the solutions in one solver and validate them in the other solver, since there are usually infinitely many solutions.
- Linear solvers use rational arithmetic and ICP uses floating point arithmetic. Efficient passing of values between the two solvers can compromise the guaranteed numerical error bounds in ICP. (See Example 2).

In this paper, we propose methods that tackle these problems. The main idea is to design an "abstraction refinement" loop between the linear and nonlinear solving stages: We use the ICP solver to search for interval solutions of the nonlinear constraints, and use the LRA solver to validate the solutions and incrementally provide more constraints to the ICP solver for refining the search space. The difficulty lies in devising procedures that efficiently communicate solutions between the linear and nonlinear solving stages without compromising numerical correctness guarantees. Our main contributions are:

- We devise procedures that separate linear and nonlinear solving in a DPLL(T) framework to enhance efficiency in solving QFNRA problems.
- We give precise definitions of correctness guarantees of ICP procedures, named δ-completeness, as used in decision problems. We show that the devised separation between linear and nonlinear solving preserves such correctness guarantees.
- We describe how to exploit ICP in assertion and learning procedures in DPLL(T) to further enhance efficiency.

The paper is organized as follows: In Section II we briefly review ICP, DPLL(T), and LRA solvers; in Section III, we show the detailed design of the checking procedures; in Section IV, we discuss correctness guarantees of ICP in decision problems; in Section V, we further describe the design of the assertion and learning procedures. We show experimental results and conclusions in Section VI and VII.

II. Background

A. Interval Constraint Propagation

The method of ICP ([15], [6]) combines interval analysis and constraint solving techniques for solving systems of real equalities and inequalities. Given a set of real constraints and interval bounds on their variables, ICP successively refines an interval over-approximation of its solution set by narrowing down the possible value ranges for each variable. ICP either detects the unsatisfiability of a constraint set when the interval assignment on some variable is narrowed to the empty set, or returns interval assignments for the variables that tightly over-approximate the solution set, satisfying some preset precision requirement. (See Fig 1.) We will only be concerned with elementary real arithmetic in this paper. We first use a simple example to show how ICP works.

Example 1. *Consider the constraint set* $\{x = y, y = x^2\}$*.*

i) Suppose $I_0^x = [1,4], I_0^y = [1,5]$ *are the initial intervals for* x *and* y*. ICP approaches the solution to the constraint set in the following way:*

Step 1. Since the initial interval of y *is* $I_0^y = [1,5]$*, to satisfy the constraint* $y = x^2$*, the value of* x *has to lie within the range of* $\pm\sqrt{I_0^y}$*, which is* $[-\sqrt{5},-1] \cup [1,\sqrt{5}]$*. Taking the intersection of* $[-\sqrt{5},-1] \cup [1,\sqrt{5}]$ *and the initial interval* $[1,4]$ *on* x*, we can narrow down the interval of* x *to* $I_1^x = [1,\sqrt{5}]$*;*

Step 2. Given $I_1^x = [1,\sqrt{5}]$ *and the constraint* $x = y$*, the interval on* y *can not be wider than* $[1,\sqrt{5}]$*. That gives* $I_1^y = I_0^y \cap [1,\sqrt{5}] = [1,\sqrt{5}]$*;*

Step 3. Given I_1^y*, we can further narrow down the interval on x, by maintaining its consistency with* $x = \pm\sqrt{y}$*, and obtain* $I_2^x = I_0^x \cap \sqrt{I_1^y} = [1,\sqrt[4]{5}]$*.*

Iterating this process, we have two sequences of intervals that approach the exact solution $x = 1, y = 1$*:*

$I^x : [1,4] \to [1,\sqrt{5}] \to [1,\sqrt[4]{5}] \to [1,\sqrt[8]{5}] \to \cdots \to [1,1]$

$I^y : [1,5] \to [1,\sqrt{5}] \to [1,\sqrt[4]{5}] \to [1,\sqrt[8]{5}] \to \cdots \to [1,1]$

ii) On the other hand, ICP detects unsatisfiability of the constraint set over intervals $I_0^x = [1.5,4]$ *and* $I_0^y = [1,4]$ *easily:*

$I^x : [1.5,4] \to [1.5,4] \cap [1,\sqrt{4}] \to [1.5,2] \to [1.5,2] \cap [\sqrt{1.5},\sqrt{2}] \to \emptyset$

$I^y : [1,4] \to [1,4] \cap [1.5,2] \to [1.5,2] \to [1.5,2] \cap \emptyset \to \emptyset$

Note that ICP implements floating point arithmetic, therefore all the irrational boundaries are relaxed by decimal numbers in practice. □

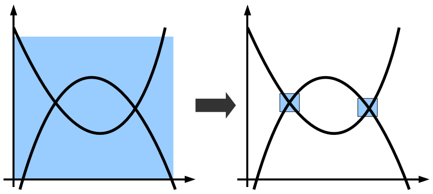

Fig. 1: Contraction of initial intervals to solution boxes

During the interval narrowing process, ICP can reach a fixed-point before the precision requirement is satisfied. In that case, ICP takes a *splitting step*, and recursively contracts the sub-intervals. This framework for solving nonlinear constraints is called the *branch-and-prune* approach [15].

We give the following formal definitions that will be referred to in the following sections. Let n be the number of variables and $\mathcal{I} = \{[a,b] : a,b \in \mathbb{R}\}$ the set of all intervals over $\mathbb{R}$. An n-ary constraint σ is a relation defined by equalities and inequalities over $\mathbb{R}$, i.e., $\sigma \subseteq \mathbb{R}^n$.

Definition 1. *Let* $\sigma \subseteq \mathbb{R}^n$ *be a constraint,* $\vec{I} \in \mathcal{I}^n$ *an interval vector whose i-th projection is written as* I_i*, i.e.,* $\vec{I} = \langle I_1, ..., I_n\rangle$*. We say* $\vec{I}$ ***over-approximates*** σ*, if for all* $(a_1,...,a_n) \in \sigma$*,* $a_i \in I_i$*.*

Definition 2. *An* ***interval contractor*** $\sharp : \mathcal{I}^n \to \mathcal{I}^n$ *is a function satisfying* $\sharp\vec{I} \subseteq \vec{I}$*. The result of multiple applications of an interval contractor on* $\vec{I}$ *is written as* $\sharp^*\vec{I}$*. A* ***contraction sequence*** *is a sequence of intervals* $S = (\vec{I}_1, ..., \vec{I}_n)$ *where* $\vec{I}_{i+1} = \sharp\vec{I}_i$*. A* ***contraction step*** *in S is defined as* $(\vec{I}, \sharp\vec{I})$ *where* $\vec{I} = \langle I_1, ..., I_i, ..., I_n\rangle$*,* $\sharp\vec{I} = \langle I_1, ..., \sharp I_i, ..., I_n\rangle$ *and*

$$\sharp I_i = I_i \cap F(I_1, ..., I_{i-1}, I_{i+1}, ..., I_n).$$

$F : \mathcal{I}^{n-1} \to \mathcal{I}$ *is an interval-arithmetic function whose graph over-approximates the constraint* σ*.*

Definition 3. *A* ***consistency condition*** $\mathcal{C} \subseteq \mathcal{I}^n \times \mathcal{I}^n$ *satisfies: for any constraint* $\sigma \subseteq \mathbb{R}^n$*, if* $\vec{I}$ *over-approximates* σ *and* $(\vec{I}, \sharp\vec{I}) \in \mathcal{C}$*, then* $\sharp\vec{I}$ *over-approximates* σ*.*

B. DPLL(T) and the Dutertre-de Moura algorithm

An SMT problem is a quantifier-free first-order formula φ with atomic formulas specified by some theory T. Most current SMT solvers use the DPLL(T) framework [19]. A DPLL(T)-based solver first uses a SAT solver on the Boolean abstraction φ^B of the formula φ. If φ^B is satisfiable, a theory solver (T-solver) is used to check whether the Boolean assignments correspond to a consistent set of *asserted theory atoms*. The T-solver should implement the following procedures:

Check() and Assert(): The Check() procedure provides the main utility of a T-solver. It takes a set of theory atoms and returns a "satisfiable"/"unsatisfiable" answer, depending on whether the set is consistent with respect to the theory T. The Assert() procedure provides a partial check for detecting early conflicts.

Learn() and Backtrack(): When the Check() or Assert() procedure detects inconsistency in a set of theory atoms, the T-solver provides *explanations* through the Learn() procedure, so that a clause can be learned for refining the search space. When inconsistency occurs, the T-solver performs efficient backtracking on the theory atoms in Backtrack().

LRA Solvers: The standard efficient algorithm for solving SMT problems with linear real arithmetic is proposed in [10], which we will refer to as the Dutertre-de Moura Algorithm. The algorithm optimizes the Simplex method for solving SMT problems by maintaining a fixed matrix for all the linear constraints so that all the operations can be conducted on simple bounds on variables. In what follows we assume that the LRA solver implements the Dutertre-de Moura algorithm.

C. Formula Preprocessing

We consider quantifier-free formulas over $\langle \mathbb{R}, \leq, +, \times \rangle$. The atomic formulas are of the form $p_i \sim c_i$, where $\sim \in \{<, \leq, >, \geq, =\}$, $c_i \in \mathbb{R}$ and p_i is a polynomial in $\mathbb{R}[\vec{x}]$.

Adopting similar preprocessing techniques as in [10], we preprocess input formulas so that a fixed set of constraints can be maintained such that the DPLL search can be done only on simple atoms of the form $x \sim c$. For any input formula, we introduce two sets of auxiliary variables: a set of nonlinear variables and a set of slack variables.

A nonlinear variable v_i is introduced when a nonlinear term t_i appears for the first time in the formula. We replace t_i by v_i and add an additional atomic formula $(t_i = v_i)$ to the original formula as a new clause.

Similarly, a slack variable s_i is introduced for each atomic formula $p_i \sim c_i$, where p_i is not a single variable. We replace p_i by s_i, and add $(p_i = s_i)$ to the original formula.

For instance, consider

$$\varphi \equiv_{df} ((x^2 + y \geq 10 \wedge x \cdot z < 5) \vee y + z > 0).$$

We introduce nonlinear and slack variables to get:

$$\underbrace{(x^2 = v_1 \wedge x \cdot z = v_2)}_{N_\varphi} \quad \wedge \quad \underbrace{(v_1 + y = s_1 \wedge y + z = s_2)}_{L_\varphi}$$
$$\wedge \quad \underbrace{((s_1 \geq 10 \wedge v_2 < 5) \vee s_2 > 0)}_{\varphi'}$$

The new formula is equi-satisfiable with the original formula. In general, after such preprocessing, any input formula φ is put into the following normal form:

$$\varphi \equiv \underbrace{\bigwedge_{i=1}^{n} \nu_i}_{N_\varphi} \wedge \underbrace{\bigwedge_{i=1}^{m} \mu_i}_{L_\varphi} \wedge \underbrace{\bigwedge_{j=1}^{p} (\bigvee_{i=1}^{q} l_{j_i})}_{\varphi'} .$$

The following notations will be used throughout the paper:

1. $V = \{x_1, ..., x_k\}$ denotes the set of all the variables appearing in φ. The set of variables appearing in any subformula ψ of φ is written as $V(\psi)$. In particular, write $V_N = \bigcup_i V(\nu_i)$ and $V_L = \bigcup_i V(\mu_i)$.

2. In N_φ, each atom ν_i is of the form $x_{i_0} = f_i(x_{i_1}, ..., x_{i_r})$ where $x_{i_j} \in V$. Note that x_{i_0} is the introduced nonlinear variable. f_i is a nonlinear function that does *not* contain addition/subtraction. We call N_φ the *nonlinear table* of φ.

3. In L_φ, each atom μ_i is of the form $\sum a_{i_j} x_{i_j} = 0$, where $a_{i_j} \in \mathbb{R}$ and $x_{i_j} \in V$. L_φ is called the *matrix* of the formula following [10].

4. In φ', each literal l_i is of the form $(x_j \sim c_i)$ or $\neg(x_j \sim c_i)$, where $x_j \in V$, $c_i \in \mathbb{R}$ and $\sim \in \{>, \geq, =\}$. The original Boolean structure in φ is now contained in φ'.

All the ν_is and μ_is are called *constraints* (nonlinear or linear, respectively), and $N_\varphi \wedge L_\varphi$ is called the *extended matrix* of the formula.

III. Interfacing Linear and Nonlinear Solving in the Check() Procedure

A. The Main Steps

As introduced in Section II-B, the Check() procedure provides the main utility of the theory solver in the DPLL(T) framework. It takes a set of asserted theory atoms and returns whether their conjunction is satisfiable in the theory T.

An intuitive way of separating linear and nonlinear solving is to have the following two stages:

1. The linear constraints are first checked for feasibility, so that linear conflicts can be detected early.

2. If no linear conflict arises, the nonlinear solver is invoked to check whether the nonlinear constraints are satisfiable *within the feasible region* defined by the linear constraints.

However, the difficulty lies in starting the second step. For checking linear feasibility, the LRA solver maintains only one point-solution throughout the solving process. That is, it stores and updates a rational number for each variable. To obtain the linear feasible region, extra computation is needed. A direct way is to use the optimization phase of the Simplex algorithm and collect optimal bounds of linear variables (their min/max values), which are used as the initial interval assignments for ICP. However, this is problematic for several reasons:

- Obtaining bounds on each variable requires solving two optimization problems involving *all* the linear constraints for *every* variable. This leads to heavy overhead.
- More importantly, the bounds on variables only constitute a box over-approximation of the linear feasible region.

After obtaining a nonlinear solution within this over-approximation, we still need to check whether this solution resides in the real feasible region. (See Fig. 2)

- Numerical errors in the optimization procedures are introduced in the decision procedure. They can compromise the correctness guarantees of ICP.

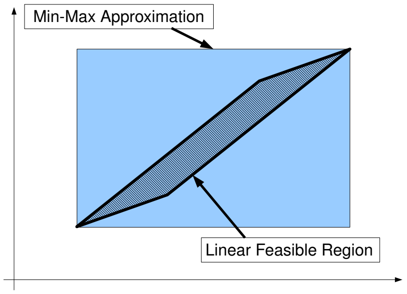

Fig. 2: Box approximations can be too coarse.

Consequently, we need more subtle interaction procedures between the linear and nonlinear solving stages.

We write the set of asserted theory atoms as Λ, i.e.,

$$\Lambda \subseteq \{x_i \sim c_i : x_i \sim c_i \text{ is a theory atom in } \varphi'\},$$

where φ' is as defined in the preprocessing step. Our Check() procedure (Fig. 3) consists of the following main steps:

Step 1. Check Linear Feasibility. (Line 2 in Fig. 3)

First, we use the LRA solver to check the satisfiability of the linear formula $L_\varphi \wedge \bigwedge \Lambda$. If the formula is unsatisfiable, there is a conflict in Λ with respect to the matrix L_φ, and Check() directly returns "*unsatisfiable*".

Step 2. Check Nonlinear Feasibility. (Line 4 in Fig. 3)

If the linear constraints are consistent, we start ICP directly on the set of nonlinear constraints; i.e., we check the satisfiability of the formula $N_\varphi \wedge \bigwedge \Lambda$. Note that after the linear solving phase in Step 1, the bounds on linear variables in Λ are already partially refined by the LRA solver [10] and we update Λ with the refined bounds. (Line 3 in Fig. 3)

If ICP determines that the nonlinear constraints are inconsistent over the initial intervals specified by Λ, the solver directly returns "*unsatisfiable*".

Step 3. Validate Interval Solutions. (L6 in Fig. 3; Fig. 4)

If ICP determines that the formula in Step 2 is satisfiable, it returns a vector $\vec{I}$ of interval assignments for all variables in V_N. Since we did not perform nonlinear checking within the linear feasible region, it is possible that the interval assignments for V_N are inconsistent with the matrix L_φ. Thus, we need to *validate* the interval solutions $\vec{I}$ with respect to the linear constraints L_φ. This validation step requires reasoning about the geometric properties of the interval solutions and linear feasibility region defined by $L_\varphi \wedge \bigwedge \Lambda$. We give detailed procedures for the validation step in Section III-B.

Step 4. Add Linear Constraints to ICP. (L7-10 in Fig. 3)

If in the previous step an interval solution $\vec{I}$ is not validated by the linear constraints, we obtain a set Σ of linear constraints (specified in Section III-B) that are violated by $\vec{I}$. Now we restart ICP and look for another solution that can in fact satisfy the linear constraints in Σ, by setting $N_\varphi := N_\varphi \wedge \bigwedge \Sigma$ and loop back to Step 2. This is further explained in Section III-C.

```
1: Procedure Check($L_\varphi, N_\varphi, \Lambda$)
2: if Linear_Feasible($L_\varphi \wedge \bigwedge \Lambda$) then
3:     $\Lambda \leftarrow$Linear_Refine($L_\varphi \wedge \bigwedge \Lambda$)
4:     while ICP_Feasible($N_\varphi \wedge \bigwedge \Lambda$) do
5:         $\vec{I} \leftarrow$ ICP_Solution($N_\varphi \wedge \bigwedge \Lambda$)
6:         $\Sigma \leftarrow Validate(\vec{I}, L_\varphi, \Lambda)$
7:         if $N_\varphi == N_\varphi \cup \Sigma$ then
8:             return satisfiable
9:         else
10:            $N_\varphi \leftarrow N_\varphi \cup \Sigma$
11:        end if
12:    end while
13: end if
14: return unsatisfiable
```

Fig. 3: Procedure Check()

```
Procedure Validate($\vec{I} = \langle \vec{l}, \vec{u} \rangle, L_\varphi, \Lambda$)
if Linear_Feasible($\bigwedge (x_i = \frac{l_i+u_i}{2}) \wedge L_\varphi \wedge \bigwedge \Lambda$) then
    $\vec{y} \leftarrow \vec{b}$  /*LRA solver returns $\vec{b}$ as the solution of $\vec{y}$*/
    for $\mu : \vec{x} \leq e_j + \vec{d}_j^{\mathrm{T}} \vec{y} \in L_\varphi$ do
        if $\vec{c}_j^{\mathrm{T}} \vec{a}_j \leq e_j + \vec{d}_j^{\mathrm{T}} \vec{b}$ is false then
            /* See Proposition 1 for the definitions */
            $\Sigma \leftarrow \Sigma \cup \mu$
        end if
    end for
else
    $\Sigma \leftarrow$ Linear_Learn($\bigwedge (x_i = \frac{l_i+u_i}{2}) \wedge L_\varphi \wedge \bigwedge \Lambda$)
end if
return $\Sigma$
```

Fig. 4: Procedure Validate()

In this way, we incrementally add linear constraints into the set of constraints considered by ICP to refine the search space. The loop terminates when ICP returns unsatisfiable on N_φ because of the newly added linear constraints, or when the LRA solver successfully validates an interval solution.

Next, we give the detailed procedures for the validation steps.

B. The Validation Procedures

1) Relations between interval solutions and the linear feasible region: Geometrically, the interval solution returned by ICP forms a hyper-box whose dimension is the number of variables considered by ICP. The location of the hyper-box with respect to the linear feasible region determines whether the interval solution for the nonlinear constraints satisfies the linear constraints. There are three possible cases (see Fig. 5 for a two-dimensional illustration):

Case 1: (Box A in Fig. 5) The hyper-box does not intersect the linear feasible region. In this case, the interval solution returned by ICP does not satisfy the linear constraints.

Case 2: (Box B in Fig. 5) The hyper-box *partially* intersects the linear feasible region. In this case, the real solution of the nonlinear constraints contained in the solution box could either reside inside or outside the linear region.

Distinguishing this case is especially important when we take into account that the LRA solver uses precise rational arithmetic. The interval assignments returned by ICP satisfy certain precision requirements and usually have many decimal digits, which can only be represented as ratios of large integers in the LRA solver. Precise large number arithmetic is costly in the LRA solver. To efficiently validate the interval solutions, we need to truncate the decimal digits. This corresponds to a further overapproximation of the intervals. For example:

Example 2. *Consider* $(y = x^2) \wedge (y - x = s) \wedge (y \geq 2 \wedge x \geq 0 \wedge s \geq 0.6)$*. In Step 2, ICP solves the formula* $(y = x^2 \wedge y \geq 2 \wedge x \geq 0)$ *and returns a solution* $x \in [1.414213562373, 1.414213567742]$ *and* $y \in [2, 2.000000015186]$*. Its rational relaxation* $x \in [14/10, 15/10]$ *and* $y \in [2, 21/10]$ *is validated, since* $y - x \geq 0.6$ *is satisfied by* $x = 1.4, y = 2$*. But the original formula is unsatisfiable, which can in fact be detected if we use ICP on the nonlinear and linear constraints together.*

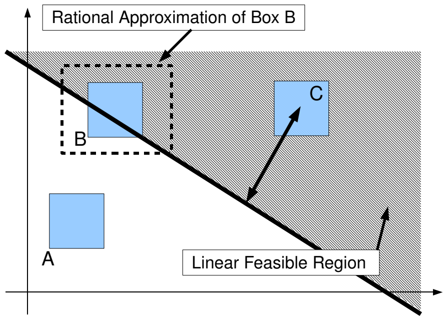

Fig. 5: Positions of hyper-boxes and the linear feasible region.

Case 3: (Box C in Fig. 5) The hyper-box completely resides in the linear feasible region. In this case, all the points in the interval solution returned by ICP satisfy the linear constraints and hence the formula should be *"satisfiable"*. To distinguish this case from Case 2, we propose the following consistency condition.

2) The Sufficient Consistency Check: We check whether *all* the points in $\vec{I}$ satisfy the linear constraints in L_φ. When that is the case, we say $\vec{I}$ is *consistent* with L_φ and accept the interval solution. This is a strong check that provides a *sufficient* condition for the existence of solutions. By enforcing it we may lose possible legitimate solutions (e.g., Box B may contain a solution that indeed resides in the linear region). This problem is handled in the refinement step (Section III-C).

We write variables contained in the nonlinear constraints as $V_N = \{x_1, ..., x_n\}$, and the variables that *only* occur in linear constraints as $V_L \setminus V_N = \{y_1, ..., y_m\}$.

Definition 4. *Let* $\vec{I} : \langle [l_1, u_1], \ldots, [l_n, u_n] \rangle$ *be an interval solution for variables in* V_N*. We write* $\vec{I} : [\vec{l}, \vec{u}]$*,* $\vec{x} = (x_1, ..., x_n)$*,* $\vec{y} = (y_1, ..., y_m)$*. We say* $\vec{I}$ *is* ***consistent*** *with the matrix* L_φ*, if*

$$\exists\, \vec{y}\, \forall\, \vec{x}\ \left[(\vec{x} \in \vec{I}) \ \rightarrow\ (L_\varphi \wedge \bigwedge \Lambda)\right] \qquad \cdots (\star)$$

is true, where $\vec{x} \in \vec{I} =_{df} \bigwedge_{x_i \in V_N} (l_i \leq x_i \wedge x_i \leq u_i)$*. Note that* $L_\varphi \wedge \bigwedge \Lambda$ *is a formula in both* $\vec{x}$ *and* $\vec{y}$*.*

This condition states that, for an interval solution $\vec{I}$ to be consistent with the linear constraints, there must be a feasible point solution $\vec{b}$ for the remaining linear variables $\vec{y}$ such that for all the points $\vec{a} \in \vec{I}$, $(\vec{a}, \vec{b})$ satisfies the linear constraints. This is a direct formulation of Case 3.

3) The Validation Steps: We propose the following procedures for validating interval solution with the LRA solver (shown in Fig. 4).

Step 3.1. (Line 2-3, Fig. 4) First, we check whether the center of the hyper-box $\vec{I}$ resides in the linear feasible region (in fact it can be an arbitrary point of the box), by checking whether the linear formula:

$$\bigwedge_{x_i \in V_N} (x_i = \frac{l_i + u_i}{2}) \wedge L_\varphi \wedge \bigwedge \Lambda$$

is satisfiable. This can be done by the LRA solver.

If this formula is unsatisfiable, we know that Condition $(\star)$ is violated, since the center of $\vec{I}$ lies outside linear feasible region. We can obtain a set of violated linear constraints provided by the LRA solver (Line 11, Fig. 4). This is further explained in Section III-C.

If it is satisfiable, the Dutertre-de Moura Algorithm returns an exact point solution $\vec{b}$ for $\vec{y}$. (Line 3, Fig. 4)

Step 3.2. (Line 4-7, Fig. 4) Next we need to ensure that, after the remaining linear variables $\vec{y}$ are assigned $\vec{b}$, the interval box $\vec{I}$ resides "far away" from the boundaries of the linear feasible region.

Since $L_\varphi \wedge \bigwedge \Lambda$ only contains linear constraints, it can be written as the intersection of k half spaces:

$$L_\varphi \wedge \bigwedge \Lambda \equiv \bigwedge_{j=1}^{k} \vec{c}_j^{\,\mathrm{T}} \vec{x} \leq e_j + \vec{d}_j^{\,\mathrm{T}} \vec{y}.$$

where $\vec{c}_j = (c_{j1}, ..., c_{jn})$ and $\vec{d}_j = (d_{j1}, ..., d_{jm})$.

First, we make the observation that the maximum of each $\vec{c}_j^{\,\mathrm{T}} \vec{x}$ is obtained when the x variables take the min or max values in their intervals depending on their coefficients:

Lemma 1. *The solution to the linear program*

$$\max\ \vec{c}_j^{\,\mathrm{T}} \vec{x} \ \ \textit{with respect to} \ \ \vec{x} \in \vec{I} : [\vec{l}, \vec{u}]$$

is given by $\vec{x} = (a_1, ..., a_n)$*, where* $a_i = l_i$ *when* $c_{ji} \leq 0$ *and* $a_i = u_i$ *otherwise.*

Further, we know that the universal statement in the consistency condition is satisfied, if the max value of $\vec{c}^{\,\mathrm{T}} \vec{x}$ is bounded by the linear constraints $e_j + \vec{d}_j^{\,\mathrm{T}} \vec{y}$. That is:

Proposition 1. *The assertion*

$$\forall\ \vec{x}.\ ((\vec{x} \in \vec{I}) \ \rightarrow\ \vec{c}_j^{\,\mathrm{T}} \vec{x} \leq e_j + \vec{d}_j^{\,\mathrm{T}} \vec{y})$$

holds for $\vec{y} = \vec{b}$ *iff*

$$\vec{c}_j^{\,\mathrm{T}} \vec{a}_j \leq e_j + \vec{d}_j^{\,\mathrm{T}} \vec{b},$$

wherein $\vec{a}_j = (a_{j1}, ..., a_{jn})$ satisfying: $a_{ji} = l_i$ when $c_{ji} \leq 0$, and $a_{ji} = u_i$ otherwise.

The condition in Proposition 1 can be verified by simple calculations: we only need to plug the values of $\vec{x} = \vec{a}_j$ and $\vec{y} = \vec{b}$ in each linear constraint, and check whether the constraint is satisfied.

To summarize, we use the LRA solver to search for a candidate solution $\vec{b}$ for the linear variables $\vec{y}$, and verify if the strong consistency condition $(\star)$ holds when $\vec{y} = \vec{b}$, using Proposition 1. If Condition $(\star)$ is verified, we return "satisfiable".

Again, Condition $(\star)$ and Proposition 1 provide a sufficient condition for the consistency of $\vec{I}$ and L_φ, which may refute legitimate solution boxes. This is compensated, because we use the strong condition to learn the violated linear constraints instead of directly refuting boxes. This is further explained in the next section.

C. Refinement of ICP Search Using Linear Constraints

In the validation steps, there are two places where we can detect that an interval solution has violated linear constraints:

- In Step 3.1, the linear formula is detected unsatisfiable by the LRA solver. In this case, we use the learning procedure in the LRA solver that returns a set of linear constraints.
- In Step 3.2, the condition in Proposition 1 can fail for a set of linear constraints. These are the constraints that the box solution does not completely satisfy.

In both cases, we have a set of linear constraints which we write as Σ. We then add Σ to N_φ and restart the ICP search on the updated N_φ. Now, the new interval solution obtained by the updated N_φ should not violate Σ modulo numerical errors in ICP, since it was obtained by ICP under the constraints in Σ.

Here, a tricky problem is that ICP allows numerical error (up to its precision bound). It is possible that even after Σ is added to ICP, the interval solution $\vec{I}$ that ICP returns may still violate Σ in terms of precise arithmetic. In such cases, the linear solver and the ICP solver disagree on the same set of constraints: Namely, ICP decides that $\vec{I}$ satisfies Σ up to its error bound, whereas the linear solver can decide that $\vec{I}$ is not consistent with Σ since it is not validated using precise arithmetic. When this happens, the same set Σ can be repeatedly violated and the refinement algorithm may loop forever without making progress. To avoid this problem, we pose the requirement that the added Σ should not be already contained in N_φ. Otherwise, we directly return "satisfiable" (Line 6 and 7 in Fig. 3). We will show in the next section that this decision preserves the correctness guarantees of ICP.

IV. CORRECTNESS GUARANTEES

Originally ICP is used in solving systems of nonlinear equalities/inequalities over real numbers. Thus, the notion of correctness of ICP is not directly formulated for the use in decision procedures. A well-known property [15] of ICP is that when a system S of real equalities and inequalities has a solution, ICP always returns an interval solution $\vec{I}$ of S. In deciding QFNRA problems, this property of ICP implies that when a system is satisfiable, ICP always returns "satisfiable". In other words, when ICP returns "unsatisfiable", the system must be unsatisfiable.

Conversely, if whenever ICP returns "satisfiable" the system is also satisfiable, we would have a sound and complete solver[1]. This can not be guaranteed by ICP because of its use of finite-precision arithmetic. In other words, the "satisfiable" answers from ICP can not always be trusted. In the design of HySAT [11], posterior validation procedures of the interval solutions are applied, and the solver can return "unknown" when a solution is not validated.

Similar validation procedures can be straightforwardly adopted in our solver. However, in what follows we aim to make clear the exact meanings of the answers returned by ICP algorithms in the context of decision problems. In fact, we will show that ICP does guarantee a moderately relaxed notion of soundness and completeness that can indeed prove useful for certain verification tasks.

Informally, when ICP returns "satisfiable" for a set S of theory atoms, it must be one of the following two cases:

- S is satisfiable.
- S is unsatisfiable, but if some constant terms in S are changed *slightly*, S will become satisfiable.

Contrapositively, if a system S remains unsatisfiable under small perturbations on its constant terms, ICP indeed returns that S is "unsatisfiable". This notion (as a special case of the formulation in [21]) is made precise in the following definition.

Definition 5 (δ-robustness)**.** *Let S be a system of equalities $\bigwedge_{i=1}^k f_i = 0$, where $f_i \in \mathbb{R}[\vec{x}]$, and $x_i \in I_i$ where $I_i \subseteq \mathbb{R}$ are intervals. Let $\delta \in \mathbb{R}^+$ be a positive real number.*

*S is called **δ-robustly unsatisfiable** if for any choice of $\vec{c} = (c_1, ..., c_k)$ where $|c_i| \leq \delta$, $\bigwedge_{i=1}^k f_i = c_i$ remains unsatisfiable. Each $\vec{c}$ is called a **perturbation** on S.*

We write the system perturbed by $\vec{c}$ as $S^{\vec{c}}$. Note that we only considered systems of equalities, because inequalities can be turned into equalities by introducing new bounded variables. The following example illustrates the definition.

Example 3. *Consider the system $S : y = x^2 \wedge y = -0.01$. S is unsatisfiable. If we set $\delta_1 = 0.1$, then there exists a perturbation $c = 0.01 < \delta_1$ such that $S^{(0,c)} : y = x^2 \wedge y = 0$ is satisfiable. However, if we set $\delta_2 = 0.001$, then there does not exist $\vec{c}$ that can make $S^{\vec{c}}$ satisfiable with $|c_i| \leq \delta_2$. Hence, we say S is δ_2-robustly unsatisfiable.* □

The bound δ of "undetectable perturbations" corresponds to the error bound of ICP. It can be made very small in practice (e.g., 10^{-6}). To be precise, we have the following theorem:

[1] The notion of soundness and completeness have quite different, although related, definitions in different communities. We will give clear definitions when a formal notion is needed (such as δ-completeness). Informally, we will only use "sound and complete" together to avoid mentioning their separate meanings that may cause confusion.

Theorem 1. *Let S be a system of real equalities and inequalities. Let δ be the preset error bound of ICP. If for any $\vec{c}$ satisfying $|c_i| \leq \delta$, $S^{\vec{c}}$ is unsatisfiable, then ICP returns "unsatisfiable" on S.*

Proof: First, note that we only need to consider systems of equalities. This is because by introducing a new variable, an inequality $f(\vec{x}) > c$ can be turned into an equality $f(\vec{x}) = y$ with the interval bound $y \in (c, +\infty)$.

Now, let $S : \bigwedge_{i=1}^{k} f_i(\vec{x}) = 0$ be a system of equalities, where the variables are bounded by the initial interval bounds $\vec{x} \in \vec{I_0}$, and $f_i \in \mathbb{R}[\vec{x}]$ are polynomials.

Suppose S is decided as satisfiable by ICP. ICP returns an interval solution $I_{\vec{x}}$ for $\vec{x}$. The δ error bound of ICP ensures that:

$$\exists \vec{x} \in I_{\vec{x}}\ [\bigwedge_{i=1}^{k} |f_i(\vec{x})| \leq \delta].$$

Let $\vec{a}$ be the witness to the above formula. We then have

$$(f_1(\vec{a}) = c_1 \wedge c_1 \leq \delta)\ \wedge ... \wedge\ (f_k(\vec{a}) = c_k \wedge c_k \leq \delta).$$

Consequently, $(c_1, ..., c_k)$ is indeed a perturbation vector that makes $S^{(c_1,...,c_k)} : \bigwedge_{i=1}^{k} f_i(\vec{x}) = c_i$ satisfiable with the solution $\vec{a}$. As a result, S is not δ-robustly unsatisfiable, which contradicts the assumption. ■

This property ensures that ICP is not just a partial heuristic for nonlinear problems, but satisfies a "numerically relaxed" notion of completeness, which we call **δ-completeness**:

- *If S is satisfiable, then ICP returns "satisfiable".*
- *If S is δ-robustly unsatisfiable, then ICP returns "unsatisfiable".*

Consequently, the answer of ICP can only be wrong on systems that are unsatisfiable but not δ-robustly unsatisfiable, in which case ICP returns "satisfiable". We can say such systems are "fragilely unsatisfiable".

In practice, it can be advantageous to detect such fragile systems. In bounded model checking, an "unsatisfiable" answer of an SMT formula means that the represented system is "safe" (a target state can not be reached). Thus, fragilely unsatisfiable systems can become unsafe under small numerical perturbations. In the standard sense, a fragilely unsatisfiable formula should be decided as "unsatisfiable" by a complete solver, and such fragility will be left undetected. Instead, ICP categorizes such fragilely unsatisfiable systems as "satisfiable". Moreover, ICP returns a solution. Note that this solution is spurious for the unperturbed system, but is informative of the possible problem of the system under small perturbations. On the other hand, ICP returns "unsatisfiable" on a system *if and only if* the system is δ-robustly safe. The error bound δ of ICP can also be changed to allow different levels of perturbations in the system.

Our checking and validation procedures are devised to preserve such correctness guarantees of ICP. Formally, we have the following theorem.

Theorem 2 (δ-completeness of Check()). *Let φ be the preprocessed input formula for Check(), and δ the error bound of ICP. If φ is satisfiable, then the Check() procedure returns "satisfiable". If φ is δ-robustly unsatisfiable, then the Check() procedure returns "unsatisfiable".*

A detailed proof of the theorem is contained in our extended technical report [13].

As a technical detail, we need to mention that the preprocessing procedure may change the actual δ in the robustness claims. The reason is that when we preprocess a formula φ to φ', new variables are introduced for compound terms, and new constants are used. Perturbations allowed on the new constants may accumulate in φ'. For instance, $x^2 = 1 \wedge x = 0$ is robustly unsatisfiable for $\delta = 1/2$. But when it is preprocessed to $x^2 - h = 0 \wedge h = 1 \wedge x = 0$, the perturbations on the first two atoms can be added, and in effect the formula is no longer $1/2$-robustly unsatisfiable ($x^2 - h = -1/2 \wedge h = 1/2 \wedge x = 0$ is satisfiable). Note that the new formula is still $1/3$-robustly unsatisfiable. The change of δ is solely determined by the number of the new variables introduced in preprocessing. In practice, when the exact error bound is needed, a new δ' can be calculated for the robustness claims that we make for the original formula. As is usually the case, the error bound is small enough (e.g. 10^{-6}) such that δ' and δ are of the same order of magnitude.

V. Assertion and Learning Procedures

In a standard DPLL(T) framework, the theory solver provides additional methods that facilitate the main checking procedures to enhance efficiency. First, a partial check named Assert() is used to prune the search space before the complete Check() procedure. Second, when conflicts are detected by the checking procedures, the theory solver uses a Learn() procedure to provide explanations for the conflicts. Such explanations consist of theory atoms in the original formula, which are added to the original formula as "learned clauses". Third, when conflicts are detected, the theory solver should backtrack to a previous consistent set of theory atoms, using the Backtrack() procedure.

In this section, we briefly describe how these additional methods can be designed when the interval methods in ICP are used in the checking procedures. A complete description of the procedures requires references to more details of ICP, which can be found in our extended technical report [13].

A. Interval Contraction in Assert()

In the DPLL(T) framework, besides the Check() procedure, an Assert() procedure is used to provide a partial check of the asserted theory atoms [10]. We use interval contraction (Definition 2) to detect early conflicts in Assert() in the following way:

In each call to Assert(), a new atom $x \sim c$ is added to the set Λ of asserted theory atoms. First, the interval assignment on x is updated by the new atom $x \sim c$. Then, Assert() contracts the interval assignment $\vec{I}$ for all the variables with respect to the linear and nonlinear constraints. That is, it takes $\vec{I}$ as input, and outputs a new vector of intervals $\vec{I'}$, such that $(\vec{I}, \vec{I'})$ is a valid contraction step (Definition 2) preserving the consistency conditions (Definition 3). If $\vec{I}'$ becomes empty, it represents an early conflict in Λ. Otherwise, Assert() returns the contracted intervals as the updated $\vec{I}$.

B. Generating Explanations and Backtracking

1) Generating Explanations: As described in Section II-A, ICP returns "unsatisfiable" when the interval assignment on some variable x is contracted to the empty set. When this happens, we need to recover the set of atoms that has contributed to the contraction of intervals on the variable x. This can be done by keeping track of the contraction steps.

Let x be a variable in φ, and l a theory atom of the form $y \sim c$ in φ. For convenience we can write l as $y \in I_c^y$. Suppose x has a contraction sequence $S_x = (I_1^x, ..., I_n^x)$. We define:

Definition 6. *The theory atom l is called a **contributing atom** for x, if there exists a contraction step (I_i^x, I_{i+1}^x) in S_x, satisfying that $I_{i+1}^x = I_i^x \cap F(\vec{I})$, where I_c^y appears in $\vec{I}$.*

The **contributing atom list** for a variable x is defined as $L_x = \bigwedge\{l : l \text{ is a contributing atom of } x\}$. We can prove that when the interval on x is contracted to the empty set, i.e., when $I_n^x = \emptyset$, it is sufficient to take the negation of L_x as the learned clause:

Proposition 2. *Let x be a variable in formula φ with a contraction sequence $(I_1^x, ..., I_n^x)$. Let L_x be the contributing atom list of x. Suppose $I_n^x = \emptyset$, then $N_\varphi \wedge L_\varphi \wedge L_x$ is unsatisfiable.*

A detailed proof is contained in [13].

2) Backtracking: When an inconsistent set Λ of atoms is detected by either Assert() or Check(), the solver calls the SAT solver to backtrack to a subset Λ' of Λ and assert new atoms. The theory solver assists backtracking by eliminating all the decisions based on atoms in $\Lambda \backslash \Lambda'$, and restores the solver state back to the decision level where Λ' is checked by Assert(). Since the Assert() procedure stores interval assignments during the contraction process, this is accomplished by restoring the interval assignment at that level.

VI. Experimental Results

We have implemented a prototype solver using the `realpaver` package for ICP [14] and the open-source SMT solver `opensmt` [4]. We accept benchmarks in the SMT-LIB [5] format, and have extended it to accept floating-point numbers. All experiments are conducted on a workstation with Intel(R) Xeon 2.4Ghz CPU and 6.0GB RAM running Linux.

A. Bounded Model Checking of Embedded Software

Our main target domain of application is bounded model checking of embedded software programs that contain nonlinear floating point arithmetic. The benchmarks (available online at [1]) in Table I are generated from unwinding a program that reads in an array of unknown values of bounded length, and tries to reach a target range by performing different arithmetic operations on the input values [16].

In Table I, We show the running time comparison between LRA+ICP and the HySAT/iSAT tool [3]. (`hysat-0.8.6` and its new version `isat` give roughly the same results on the benchmarks, we picked the best timings.)

D	#Vars	$\#L_\varphi$	$\#N_\varphi$	#l	Result	**LRA+ICP**	HySAT
Benchmark Set: AddArray							
6	10	3	0	1	`UNSAT`	0.06s	0.04s
8	36	10	0	1	`UNSAT`	0.09s	303.03s
31	1634	735	0	1	`UNSAT`	0.93s	mem-out
Benchmark Set: MultiArray-1							
5	10	3	20	1	`UNSAT`	0.23s	0.02s
7	30	8	28	1	`UNSAT`	0.04s	7.21s
8	121	40	32	1	`UNSAT`	0.12s	56.46s
16	817	320	64	1	`UNSAT`	0.32s	mem-out
26	1687	670	104	2	`SAT`	87.45s	mem-out
Benchmark Set: MultiArray-2							
9	208	75	36	1	`UNSAT`	0.73s	244.85s
10	295	110	40	1	`UNSAT`	0.11s	123.02s
11	382	145	44	1	`UNSAT`	0.12s	3.96s
20	1165	460	80	1	`UNSAT`	0.30s	mem-out
26	1687	670	104	2	`SAT`	65.72s	mem-out
Benchmark Set: MultiArrayFlags							
11	861	337	44	1	`UNSAT`	0.19s	mem-out
21	2131	847	84	1	`UNSAT`	0.93s	mem-out
31	3401	1357	124	1	`UNSAT`	0.65s	mem-out
51	5941	2377	204	1	`UNSAT`	26.17s	mem-out

TABLE I: LRA+ICP and HySAT on BMC Benchmarks

name	cvc3(s)	LRA+ICP	name	cvc3(s)	LRA+ICP
10u05	2.21	8.87	20revert	6.73	36.12
20u10	5.54	14.25	30u15	13.52	120.21
40f10	117.53	89.01	40f25	123.97	175.28
40f50	228.25	99.26	40f99	240.11	215.12
40m10	120.16	86.29	40m25	120.18	153.01
40m50	213.12	111.87	40m99	237.87	217.92
40s10	41.445	280.06	40s25	40.38	180.15
40s50	37.59	180.12	40s99	35.23	189.43
40u20	28.31	231.21	c40f	timeout	270.12
c40m	timeout	279.45	c40s	34.12	301.76
l40f	15.02	320.12	l40s	20.32	242.75
m40e	25.72	113.23	m40	226.21	182.12

TABLE II: LRA+ICP and CVC3 on QF_UFNRA Benchmarks

In the table, the first column ("D") is the unrolling depth of the original program. The number of variables (#Vars), linear constraints ($\#L_\varphi$), and nonlinear constraints ($\#N_\varphi$) are the ones that actually effective in the theory solver, after preprocessing is done. They can be much lower than the raw numbers appearing in the benchmark. The "#l" column is the number of iterations of the linear-nonlinear checking loop (Step 2-4 in Section III-A) that are used in obtaining the answer. "mem-out" indicates that HySAT aborted for the reason that no more memory can be allocated.

For the "`UNSAT`" instances, the linear solver detects conflicts in the linear constraints early on, and avoids solving the nonlinear constraints directly as in HySAT. For the "`SAT`" instances, the two iterations of the linear-nonlinear checking loop proceed as follows: Initially, no linear conflicts were detected, and ICP is invoked to solve the nonlinear constraints and return an interval solution. The linear solver then detects that the interval solutions violate a set of linear constraints, which are added to the constraints considered by ICP (Line 6 in Fig. 3). This concludes the first iteration. In the second iteration, ICP solves the expanded set of constraints and return

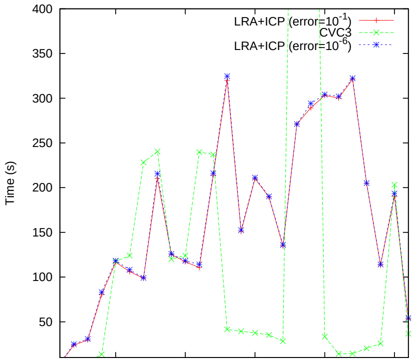

Fig. 6: LRA+ICP and CVC3 on QF_UFNRA Benchmarks

a new interval solution. Then the linear solver detects no further conflict and return "SAT" as the final answer. This concludes the second iteration.

We see that by separating the linear and nonlinear solving stages, we can exploit the highly efficient linear solver for solving the linear constraints and invoke the nonlinear solving capacity of ICP only when needed. The proposed refinement loop ensures that the correctness guarantees of ICP are preserved under such separation of linear and nonlinear solving.

B. QFNRA Problems from SMT-LIB

We have obtained results on QF_UFNRA benchmarks on SMT-LIB [5]. So far the only solver that solves the same set of benchmarks is CVC3 [2]. (The HySAT solver uses a special format. CVC3 does not accept floating point numbers in the previous set of benchmarks.)

In Table II we compare the LRA+ICP solver with CVC3. The data are plotted in Fig 6. (The timeout limit is 1000s.) The timing result is mixed. Note that our solver ensures δ-completeness and does not have specific heuristics. Consequently, our solver performs rather uniformly on all the benchmarks, whereas CVC3 can be much faster or slower on some of them. (We are not aware of the solving strategy in CVC3.) To evaluate the influence of the error bound δ on the speed of the solver, we have set it to different values $\delta_1 = 10^{-1}$ and $\delta_2 = 10^{-6}$. However, the difference is not significant on this set of benchmarks. The reason for this may be that the nonlinear constraints in the benchmarks are all of the simple form "$x = yz$" with few shared variables.

VII. Conclusion

We have proposed a novel integration of interval constraint propagation with SMT solvers for linear real arithmetic to decide nonlinear real arithmetic problems. It separates linear and nonlinear solving stages, and we showed that the proposed methods preserve the correctness guarantees of ICP. Experimental results show that such separation is useful for enhancing efficiency. We envision that the use of numerical methods with correctness guarantees such as ICP can lead to more practical ways of handling nonlinear decision problems. Further directions involve developing heuristics for different systems with specific types of nonlinear constraints and extend the current results to transcendental functions.

Acknowledgment

We thank Professor Martin Fränzle and the HySAT project group for their help and important suggestions (Professor Fränzle pointed out the problem that the robustness parameters can be changed due to preprocessing), and the anonymous referees for their valuable comments. We thank Professor Clark Barrett for his patient help on CVC3.

References

[1] http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php, Item 7.
[2] CVC3. `http://cs.nyu.edu/acsys/cvc3/index.html`.
[3] Hysat. `http://hysat.informatik.uni-oldenburg.de`.
[4] Opensmt. `http://code.google.com/p/opensmt/`.
[5] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
[6] F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16. Elsevier, 2006.
[7] L. Bordeaux, Y. Hamadi, and M. Y. Vardi. An analysis of slow convergence in interval propagation. In *CP*, pages 790–797, 2007.
[8] C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. Solving non-linear polynomial arithmetic via sat modulo linear arithmetic. In *CADE*, pages 294–305, 2009.
[9] C. W. Brown and J. H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC-2007*.
[10] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV-2006*.
[11] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.
[12] M. K. Ganai and F. Ivančić. Efficient decision procedure for non-linear arithmetic constraints using cordic. In *Formal Methods in Computer Aided Design (FMCAD)*, 2009.
[13] S. Gao, M. Ganai, F. Ivančić, A. Gupta, S. Sankaranarayanan, and E. M. Clarke. Integrating icp and lra solvers for deciding nonlinear real arithmetic problems. Technical report, "`http://www.cs.cmu.edu/∼sicung/papers/fmcad10.html`".
[14] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.
[15] P. V. Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2):797–827, 1997.
[16] F. Ivančić, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Numerical stability analysis of floating-point computations using software model checking. In *MEMOCODE 2010*.
[17] L. Jaulin. Localization of an underwater robot using interval constraint propagation. In *CP*, pages 244–255, 2006.
[18] L. Jaulin, M. Kieffer, O. Didrit, and È. Walter. *Applied Interval Analysis with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag, London, 2001.
[19] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
[20] A. Platzer, J.-D. Quesel, and P. Rümmer. Real world verification. In *CADE*, pages 485–501, 2009.
[21] S. Ratschan. Quantified constraints under perturbation. *J. Symb. Comput.*, 33(4):493–505, 2002.