Interval Constraint Propagation

Seminar in Satisfiability Checking

Ömer Sali
Supervision: Johanna Nellen

Abstract

Numerous problems in the areas of soft- and hardware verification, optimisation and planning can be formalised by quantifier-free formulas over the theory of nonlinear real arithmetic. The runtime of existing decision procedures based on algorithms such as cylindrical algebraic decomposition depends heavily on the size of the underlying solution search space of a given formula. Interval constraint propagation (ICP) serves as a method for reducing this search space prior to a call of a complete decision procedure. Applied to linear constraints, ICP can suffer from the slow convergence problem, where the use of dedicated linear real arithmetic (LRA) solvers would be superior. In this paper, we give a full description of ICP and present an approach for separating the linear and nonlinear solving stages. This results in an integration of ICP with LRA solvers to efficiently decide nonlinear real arithmetic problems.

1 Introduction

Propositional logic is well-suited for the verification of logic programs or the bounded model checking of discrete systems such as digital controllers. Its importance led to the development of highly efficient decision procedures like the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm to determine the satisfiability of Boolean formulas (see [2, Chapter 2]). Other inherently continuous problems in the area of optimisation require the expressiveness of theories. Therefore, propositional logic is extended with theory constraints to so called *satisfiability modulo theories* (SMT). The focus of this paper lies on SMT formulas that are Boolean combinations of polynomial constraints with real-valued variables:

Definition 1.1 *The syntax of a formula in the quantifier-free fragment of the nonlinear real arithmetic (QFNRA) is defined by the following grammar:*

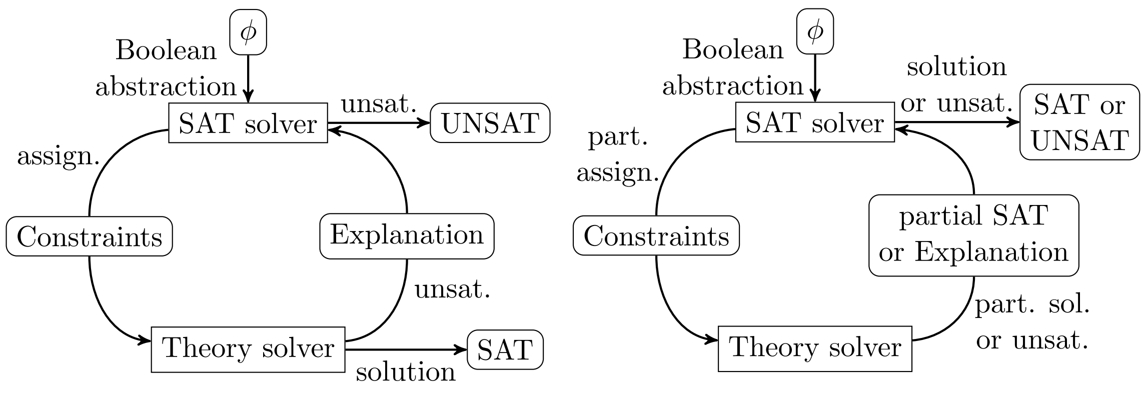
$$\begin{aligned} formula &::= constraint \mid (formula \land formula) \mid (formula \lor formula) \mid (\neg formula) \\ constraint &::= term < term \mid term \leq term \mid term = term \mid term \geq term \mid term > term \\ term &::= variable \mid constant \mid term + term \mid term \cdot term \end{aligned}$$

where constant $\in \mathbb{R}$*, and variable* $\in \{x_1, x_2, \ldots\}$ *with values from the domain* $\mathbb{R}$*.*

To clarify the structure of QFNRA formulas, consider for example

$$\phi := ((\sigma_1 : x_1^2 + x_2 \geq 10 \land \sigma_2 : x_1 \cdot x_3^3 \leq 5) \lor \sigma_3 : x_2 + 2 \cdot x_3 = 0),$$

which consists of three constraints $\sigma_1, \sigma_2, \sigma_3$ with real-valued variables x_1, x_2, x_3. The only difference between linear and nonlinear formulas is, that the multiplication of variables instead of just variables and constants is allowed. But while highly efficient algorithms like Simplex or the Ellipsoid method exist for deciding *linear real arithmetic* (QFLRA), efficient decision procedures for QFNRA are rather sparse. Existing complete algorithms like cylindrical algebraic decomposition (CAD) show a double-exponential runtime even on practical problems. The runtime depends especially on the size of the underlying solution



(a) The full lazy approach

(b) The less lazy approach

Figure 1: The basic scheme of DPLL-based SMT solving

search space. Interval constraint propagation (ICP) is an efficient numerical method for finding tight interval over-approximations of solution sets of QFNRA formulas prior to a call of a complete decision procedure. It uses the given set of constraints for deductions that lead to a contraction of the initial search space without excluding any solutions. However, a problem with ICP is that it handles both linear and nonlinear constraints for inference. Applied to linear constraints, it can suffer from the slow convergence problem (see Example 3.3 on page 5). As there already exist optimised algorithms for deciding linear arithmetic problems, solving all constraints in ICP is suboptimal. We tackle this problem by separating the linear and nonlinear solving stages. We use ICP to search for interval solutions of the nonlinear constraints, and a LRA solver to validate the solutions and incrementally provide more constraints to ICP for the search space refinement.

2 Preliminaries

Several tools for deciding the satisfiability of SMT formulas over a quantifier-free first-order theory T rely on the DPLL(T) framework: They combine a Boolean satisfiability solver based on the DPLL procedure to resolve the Boolean structure of a given formula, and a dedicated theory solver capable of verifying the consistency of conjunctions of theory constraints. In what follows, we have a closer look on the DPLL(QFNRA) approach.

A given QFNRA formula ϕ is first transformed into an equisatisfiable formula ϕ^{CNF} in conjunctive normal form. This can be done efficiently by using Tseitin's encoding to get

$$\phi^{\text{CNF}} = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} \ell_{ij} \text{ with } \ell_{ij} \in \{\sigma_{ij}, \neg\sigma_{ij}\} \text{ for QFNRA constraints } \sigma_{ij}.$$

Next, the negations in negative literals $l_{ij} = \neg\sigma_{ij}$ with a constraint $\sigma_{ij} : p_{ij} \sim q_{ij}$ and a relation $\sim \in \{<, \leq, =, \geq, >\}$ are eliminated by pushing them to the theory constraints: $<$ and $\leq$ get replaced with $\geq$ and $>$ and vice versa. Since no inequalities are allowed as relations in Definition 1.1 of QFNRA constraints, equalities like $p_{ij} = q_{ij}$ are negated by replacing them with the clause $p_{ij} < q_{ij} \vee p_{ij} > q_{ij}$, which does not affect the CNF structure. In the following, we assume that this lightweight transformation was already done and that ϕ^{CNF} consists of positive literals only. From this, the Boolean abstraction ϕ^{B} is constructed by introducing a fresh Boolean variable e_{ij} for every new constraint σ_{ij}

and keeping the Boolean skeleton intact, which gives

$$\phi^{\mathrm{B}} = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{l_i} e_{ij}, \text{ where } e_{ij} \text{ replaces the constraint } \sigma_{ij}.$$

A DPLL-based SAT solver now systematically tries to find assignments for the Boolean skeleton ϕ^{B} (see Figure 1a). For every found model $\alpha \models \phi^{\mathrm{B}}$, the corresponding set of constraints $\Sigma = \{\sigma_{ij} \mid \alpha \models e_{ij}\}$ is handed over to the theory solver and checked for consistency. Recall that the Boolean abstraction ϕ^{B} does not contain any negations and is therefore a monotone formula. Hence, the original formula ϕ must be satisfiable if and only if Σ is consistent. If the theory solver fails to find a solution of the given constraint set Σ, it provides a preferably minimal *infeasible subset* of Σ as explanation to the SAT solver, which is used to narrow the search for feasible assignments. The formula is declared to be unsatisfiable, if the SAT solver is not able to find any further satisfying assignments.

Above, we described the progress of a DPLL-based SMT solver working in *full lazy* mode, that first finds a complete assignment for the Boolean skeleton of a formula before invoking the theory solver. A major improvement of this approach is shown in Figure 1b operating in *less lazy* mode: Here, the theory solver is called more often already for incomplete assignments. Depending on the answer of the theory solver on this partial constraint set, the SAT solver can adjust its partial solution until a complete assignment is found. This approach requires a theory solver that manages an internal state to make use of previous consistency checks. It should support *incrementality* by allowing the belated assertion of new and removal of already asserted constraints. From this list of requirements we conclude, that such a theory solver should implement the following minimal interface:

assert/remove(Constraint): These procedures realise the incrementality of the theory solver for adding/removing constraints for the next consistency check. Note, that remove() must undo all the effects of the given constraint on the entire calculation.

check(): This main function performs the consistency check over all asserted constraints and returns a SAT/UNSAT answer. In case of satisfiability a model is constructed. Otherwise, check() provides a preferably minimal infeasible subset as explanation.

For the next sections, we take a LRA solver and a complete NRA solver both implementing this interface as given and develop on top of them a new ICP module with this interface.

3 Interfacing ICP with LRA solvers

For the rest of this section, let $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ be a set of QFNRA constraints with variables $\mathrm{Var}(\Sigma) = \{x_1, \ldots, x_n\}$. By $I = [\underline{I}, \overline{I}]$ we denote a closed interval with lower bound $\underline{I} \in \mathbb{R} \cup \{-\infty\}$ and upper bound $\overline{I} \in \mathbb{R} \cup \{+\infty\}$. Here, $\pm\infty$ are of course not included in I, we merely write $[-\infty, +\infty]$ instead of $(-\infty, +\infty)$ to avoid case distinctions. Suppose that the solution set $\mathrm{Sol}(\Sigma) = \{\vec{p} \in \mathbb{R}^n \mid \vec{p} \models \Sigma\}$ of Σ is bounded by an initial interval box $I^0 \in \mathbb{I}^n$, where $\mathbb{I}$ is the set of all closed intervals as defined before. The interval I_j^0 is the domain of the variable x_j in the constraint set Σ for all $1 \leq j \leq n$. For a detailed description of the following subject, refer to [1, Sections 2–3] and [3, Chapters 2–4].

3.1 Interval Constraint Propagation

The main idea of interval constraint propagation (ICP) is to efficiently reduce the initial interval box I^0 of the constraint set Σ without losing any existing solution. ICP either

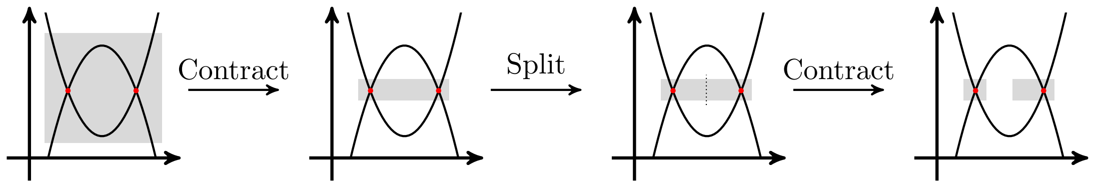


Figure 2: An exemplary contraction sequence

detects the unsatisfiability of Σ if the interval domain of some variable is narrowed to the empty set, or it returns an interval assignment for the variables that tightly over-approximates the solution set $\mathrm{Sol}(\Sigma)$, satisfying some preset precision requirement.

Example 3.1 *Consider the set $\Sigma = \{\sigma_1 : x_2 = 2x_1 + 1\}$ together with the initial interval box $I^0 = [1, 3] \times [5, 9]$. To narrow down this search space, we may argue as follows:*

- *Since the initial interval of x_2 is $I_2^0 = [5, 9]$, to satisfy the constraint σ_1, the value of x_1 has to lie within $[2, 4]$. Taking the intersection with the initial interval $I_1^0 = [1, 3]$ on x_1, we can narrow down the interval of x_1 to $I_1^1 = [2, 3]$.*
- *Since the initial interval of x_1 is $I_1^0 = [1, 3]$, to satisfy the constraint σ_1, the value of x_2 has to lie within $[3, 7]$. Taking the intersection with the initial interval $I_2^0 = [5, 9]$ on x_2, we can narrow down the interval of x_2 to $I_2^1 = [5, 7]$.*

Thus, the new interval box is given by $I^1 = [2, 3] \times [5, 7]$.

In the above example, the given constraint itself is used for an argumentation which leads to a tightening of the initial interval box. The main idea of ICP is to fully automate this reasoning process by using the given set of constraints for contraction.

Definition 3.1 *A* contractor *$C_\sigma : \mathbb{I}^n \to \mathbb{I}^n$ associated to a constraint $\sigma \in \Sigma$ is an operator such that for every interval box $I \in \mathbb{I}^n$ the two following properties are satisfied:*

Contractance: *$C_\sigma(I) \subseteq I$. Thus, the contraction never results in a wider box.*

Consistency: *$C_\sigma(I) \cap \mathrm{Sol}(\sigma) = I \cap \mathrm{Sol}(\sigma)$. Thus, no solutions are excluded.*

In practice, there are different approaches to build concrete interval contractors. Here, we use the main theory based on *interval analysis*. In order to perform the bound estimations needed in Example 3.1 systematically, interval analysis extends the arithmetic operations $\odot \in \{+, -, \cdot, \div\}$ from the domain of real numbers $\mathbb{R}$ to the set of intervals $\mathbb{I}$, such that

$$\forall I, J \in \mathbb{I} : S := \{x \odot y \mid x \in I, x \in J\} \subseteq I \odot J.$$

This property claims that the resulting interval $I \odot J$ constitutes an over-approximation of the set S. Consider the following definitions of the basic interval arithmetic operators:

Definition 3.2 *For intervals $I = [\underline{I}, \overline{I}]$ and $J = [\underline{J}, \overline{J}]$ define the* interval arithmetic

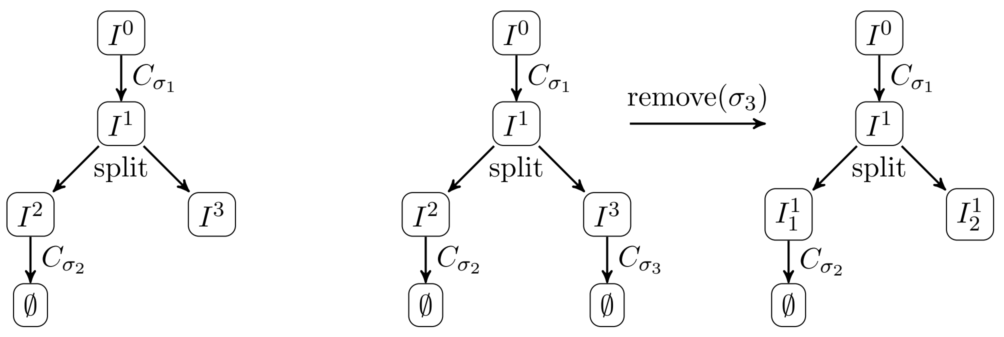
Addition: $I + J := [\underline{I} + \underline{J}, \overline{I} + \overline{J}]$ **Subtraction:** $I - J := [\underline{I} - \overline{J}, \overline{I} - \underline{J}]$

Multiplication: $I \cdot J := [\min(\underline{I} \cdot \underline{J}, \underline{I} \cdot \overline{J}, \overline{I} \cdot \underline{J}, \overline{I} \cdot \overline{J}), \max(\underline{I} \cdot \underline{J}, \underline{I} \cdot \overline{J}, \overline{I} \cdot \underline{J}, \overline{I} \cdot \overline{J})]$

Division ($0 \notin J$): *$I \div J := I \cdot (1/J)$, where $1/J := [1/\overline{J}, 1/\underline{J}]$*

We are now in a position to define interval contractors: Fix a constraint $\sigma \in \Sigma$ and an interval box $I \in \mathbb{I}^n$. If $x_j \notin \mathrm{Var}(\sigma)$, we simply let $C_\sigma(I)_j := I_j$. Otherwise, write σ as

$$\sigma : x_j \sim p_j(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n),$$



(a) An exemplary state tree **(b)** State tree pruning to remove the effect of σ_3

Figure 3: Management of the internal state with the help of a state tree

where p_j does not contain the variable x_j. Replace all variables in p_j by their interval bounds and apply interval arithmetic to calculate $J_j := p_j(I_1, \ldots, I_{j-1}, I_{j+1}, \ldots, I_n)$. Depending on the type of the relation $\sim$ in σ, we get the contracted interval as

$$C_\sigma(I)_j := \begin{cases} \text{if } \underline{I_j} \geq \overline{J_j} \text{ then } \emptyset \text{ else } [\underline{I_j}, \min(\overline{I_j}, \overline{J_j})] & , \text{if } \sigma : x_j < p_j \\ [\underline{I_j}, \min(\overline{I_j}, \overline{J_j})] & , \text{if } \sigma : x_j \leq p_j \\ [\max(\underline{I_j}, \underline{J_j}), \min(\overline{I_j}, \overline{J_j})] & , \text{if } \sigma : x_j = p_j \\ [\max(\underline{I_j}, \underline{J_j}), \overline{I_j}] & , \text{if } \sigma : x_j \geq p_j \\ \text{if } \overline{I_j} \leq \underline{J_j} \text{ then } \emptyset \text{ else } [\max(\underline{I_j}, \underline{J_j}), \overline{I_j}] & , \text{if } \sigma : x_j > p_j \end{cases} .$$

This completes the definition of the contractor C_σ. Consider again Example 3.1.

Example 3.2 *With the above notation, we again get the same result I^1 via*

- $J_1 = 0.5 \cdot ([5, 9] - 1) = [2, 4]$ *and hence* $I^1_1 = C_{\sigma_1}(I^0)_1 = [2, 4] \cap [1, 3] = [2, 3]$,
- $J_2 = 2 \cdot [1, 3] + 1 = [3, 7]$ *and hence* $I^1_2 = C_{\sigma_1}(I^0)_2 = [3, 7] \cap [5, 9] = [5, 7]$.

Unfortunately, the sequence of contractions $I^0, I^1 := C_{\sigma_{i_1}}(I^0), I^2 := C_{\sigma_{i_2}}(I^1), \ldots$ may converge to a fixed point even before a preset threshold for the desired diameter is reached (see Figure 2). To cope with this problem, we can choose a variable x_j along which an *autonomous split* of the interval box I^i is performed. This results in two new interval boxes I^{i+1} and I^{i+2} with $I^i = I^{i+1} \uplus I^{i+2}$ which can be further contracted, splitted, and searched separately for solutions. The progress of these propagation steps is kept in an internal state tree as shown in Figure 3a that constitutes the state of the ICP algorithm.

3.2 Formula preprocessing

As pointed out in the last subsection, the proper definition of contractors requires the given constraints to be solvable in every existing variable. Another major drawback of the basic ICP algorithm we have seen so far is, that its speed of convergence depends on the polynomial degree of a constraint used for contraction. Applied to linear constraints, ICP is therefore known to suffer from the *slow convergence phenomenon*:

Example 3.3 (Slow convergence) *For any number $n \in 2\mathbb{N}$ consider the constraint set $\Sigma = \{\sigma_1 : x_1 = x_2 + 1, \sigma_2 : x_2 = x_1\}$ together with the initial interval box $I^0 = [0, n] \times [0, n]$. The shortest contraction sequence of ICP to decide unsatisfiability is given by*

$$I^0 = [0, n] \times [0, n] \xrightarrow{C_{\sigma_1}} [1, n] \times [0, n-1] \xrightarrow{C_{\sigma_2}} [1, n-1] \times [1, n-1] \xrightarrow{C_{\sigma_1}}$$

$$[2, n-1] \times [1, n-2] \xrightarrow{C_{\sigma_2}} [2, n-2] \times [2, n-2] \xrightarrow{C_{\sigma_1}} \cdots [\tfrac{n}{2}, \tfrac{n}{2}] \times [\tfrac{n}{2}, \tfrac{n}{2}] \xrightarrow{C_{\sigma_1}} \emptyset$$

which consists of $n+1$ contraction steps. In contrast, the Simplex method needs at most a constant number of pivot steps regardless of the chosen number $n \in 2\mathbb{N}$.

Both problems are covered by the following preprocessing to separate the linear and nonlinear parts in a given constraint $\sigma \in \Sigma$: First, write σ in the standard form

$$\sigma : \quad \sum_{i=0}^{k} c_i \prod_{j=0}^{l_i} x_j^{e_{ij}} \sim d, \text{ where } \sim \in \{<, \leq, =, \geq, >\},\ c_i, d \in \mathbb{R}, \text{ and } e_{ij} \in \mathbb{N}.$$

For every new nonlinear monomial $m_i := \prod_{j=0}^{l_i} x_j^{e_{ij}}$ that was not considered before, replace m_i in σ by a fresh variable v_i and add the *nonlinear substitution* $m_i = v_i$ as an additional constraint. If the left hand side of the resulting *linearisation*

$$\sigma' : \quad \sum_{i=0}^{k} c_i y_i \sim d, \text{ where } y_i \in \{x_i, v_i\}$$

consists of only one term, then we denote σ' as a *bounding constraint.* From now on, we let N be the set of all nonlinear substitutions and bounds, and let L be the set of all linearisations obtained by successively preprocessing every constraint $\sigma \in \Sigma$.

Example 3.4 *The set $\Sigma := \{\sigma_1 : 2x_1^2 - x_2 = 1, \sigma_2 : x_1^2 + x_2^3 > 10\}$ yields the decomposition*

$$N = \{x_1^2 = v_1, x_2^3 = v_2\} \quad \textit{and} \quad L = \{2v_1 - x_2 = 1, v_1 + v_2 > 10\}.$$

Note, that the nonlinear constraint $x_1^2 = v_1$ is only introduced once when the nonlinear monomial x_1^2 appears in σ_1 for the first time and then reused in σ_2 for substitution.

3.3 Incrementality and backtracking

For the communication of the ICP module with a DPLL-based SAT solver in less lazy mode, we have to provide procedures for the assertion and removal of constraints.

Algorithm 3.1 Activate the contractors of the given constraint.

```
1: procedure ICP.ASSERT(Constraint $\sigma$)
2:     $(\ell, N) \leftarrow$ PREPROCESS($\sigma$)
3:     LRA.ASSERT($\ell$)
4:     NRA.ASSERT($\sigma$)
5:     ACTIVATECONTRACTORS($N$)
6:     SAVE($\sigma, (\ell, N)$)
```

Algorithm 3.2 Remove all effects of the given constraints on the entire calculation.

```
1: procedure ICP.REMOVE(Constraint $\sigma$)
2:     $(\ell, N) \leftarrow$ LOOKUPDECOMPOSITION($\sigma$)
3:     LRA.REMOVE($\ell$)
4:     NRA.REMOVE($\sigma$)
5:     DEACTIVATECONTRACTORS($\{\ell\} \cup N$)
6:     PRUNESTATETREE($\{\ell\} \cup N$)
7:     REMOVE($\sigma, (\ell, N)$)
```

Whenever a new constraint $\sigma \in \Sigma$ is asserted to the ICP module, it is preprocessed according to the last subsection to decompose it into a linearisation ℓ and a set N of nonlinear substitutions and bounds (see Algorithm 3.1, line 2). The linearisation gets asserted to the LRA module, while the original constraint σ is asserted to the complete NRA backend module (lines 3–4). Later in the consistency check function, we will query the LRA module to validate found interval boxes against these linear constraints, and the NRA backend module to search for solutions only within already contracted interval boxes. Initially, the ICP module performs contractions only using nonlinear constraints, whose contractors are activated in line 5. In case the check function later on detects the inconsistency of the asserted constraints, the collection of all constraints that were used for contraction forms an infeasible subset. But since the infeasible subset needs to be a subset of the originally asserted constraints, we must be able to retrieve the origins of these preprocessed constraints. This requires, that a mapping from original constraints (referred to as origins) to the modified constraints has to be kept (line 6).

The remove procedure (see Algorithm 3.2) withdraws almost all steps done by the assert procedure in the same way, except for the following two differences: During the consistency check, selected linear constraints are activated for contraction, which must be deactivated in addition to the nonlinear constraints (line 5). Furthermore, the internal state of the ICP module represented by its state tree must be pruned, such that all effects of the given constraint on the entire calculation are removed as well (see Figure 3b).

3.4 Interfacing ICP with the LRA solver for the consistency check

Ideally, we would like to separate the linear and nonlinear solving stages and apply efficient algorithms for linear constraints as much as possible. Since both types of constraints share many variables in nontrivial problems, a complete separation is illusory. The difficulty lies in devising a consistency checking procedure that efficiently uses the point solutions returned by an LRA solver to incrementally refine the interval boxes generated by ICP. In Algorithm 3.3, we use the ICP solver to search for interval solutions of the nonlinear constraints, and use the LRA solver afterwards to validate the found solution box.

Algorithm 3.3 Check the consistency of the asserted constraints.

```
 1: function ICP.check()
 2:     if LRA.check() then
 3:         while hasNextBox() do
 4:             $I \leftarrow$ getNextBox()
 5:             $\Sigma_{\text{conf}} \leftarrow$ validate($I$)
 6:             if $\Sigma_{\text{conf}} = \emptyset$ then
 7:                 NRA.assert($I|_{\text{Var}(\Sigma)}$)
 8:                 answer $\leftarrow$ NRA.check()
 9:                 NRA.remove($I|_{\text{Var}(\Sigma)}$)
10:                 if answer = SAT then
11:                     ICP.model $\leftarrow$ NRA.model
12:                     return SAT
13:                 else
14:                     ICP.infSubset $\overset{+}{\leftarrow}$ NRA.infSubset
15:                     discardBox($I$)
16:             else
17:                 activateContractors($\Sigma_{\text{conf}}$)
18:         ICP.infSubset $\overset{+}{\leftarrow}$ lookupOrigins(getUsedConstraints())
19:     else
20:         ICP.infSubset $\leftarrow$ lookupOrigins(LRA.infSubset)
21:     return UNSAT
```

The first step is to invoke the consistency check of the LRA solver to check the satisfiability of the linearisations L asserted so far (line 2). If these linear constraints are already inconsistent, this saves us from entering the expensive constraint propagation loop (lines 3–17). In this case, the LRA solver provides an infeasible subset of the asserted constraint set L as explanation (line 20). Since an infeasible subset of ICP has to be a subset of the original constraints Σ, this can not be returned directly. Instead, we look up the collection of all original constraints from which one of them originated from during preprocessing. This collection must also be unsatisfiable, since the preprocessing ensured the equisatisfiability of the original and the preprocessed constraints.

If the linear constraints are consistent, we start ICP directly on the set of nonlinear substitutions and bounds N asserted so far (line 3). When the ICP module finds a solution box I fulfilling the precision requirements (line 4), this box is validated against the linear constraints held by the LRA solver (line 5). This validation procedure is the key component in the integration of ICP with the LRA solver and hence discussed in the next subsection in more detail. The result of the validation is the *conflict set* Σ_{conf} of all linear constraints violating some point in I, which we activate as additional contractors (line 17). This allows ICP to incrementally refine its search space before returning the next suitable box in the next iteration of the loop. If no linear constraints are violated, we hand I over to the complete, but expensive NRA backend module (lines 7–15).

For this purpose, we temporarily assert the search box to the complete backend in addition to the already asserted original constraints Σ and invoke the consistency check (lines 7–9). Note, that I also contains superfluous intervals for nonlinear variables introduced during preprocessing, which are not needed by the NRA solver. Therefore, I is restricted to the set of variables $\text{Var}(\Sigma)$ contained in any original constraint. If a solution is found, we take the model of the NRA solver as the model of our ICP module before reporting satisfiability (lines 11–12). If the complete backend rejects the passed search box, I does not contain any solution for Σ, and we discard it to select the next search box (lines 14–15). The infeasible subset returned by the NRA solver is added to the corresponding infeasible subset of the ICP algorithm.

The constraint propagation loop terminates, if every search box that resulted from a split was rejected. In this case, the collection of all constraints which were used during the whole calculation forms an infeasible subset. The constraints that were used by the NRA solver to prove unsatisfiability were already added to the infeasible subset. The constraints used for contractions can be easily read out of the state tree. As before, we take the origins of these preprocessed constraints before reporting unsatisfiability (line 18).

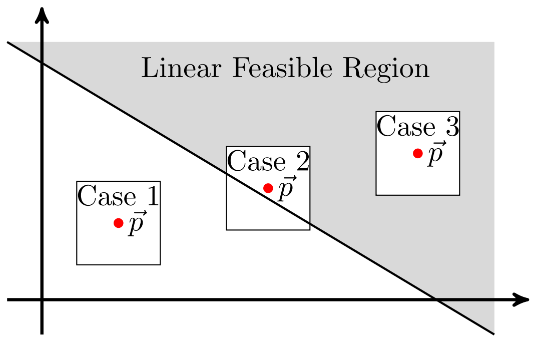
The expectation behind the presented algorithm is, that the number of linear constraints violating a found search box is rather small compared to the number of all linear constraints. But even if this assumption is not fulfilled in a worst case scenario, the stepwise structure for testing necessary conditions enhances the runtime: We start with a cheap invocation of the LRA solver, continue with the analysis of the nonlinear constraints and end up in a final inevitable call of the expensive complete NRA backend module.

3.5 Validating a solution box against the linear feasible region

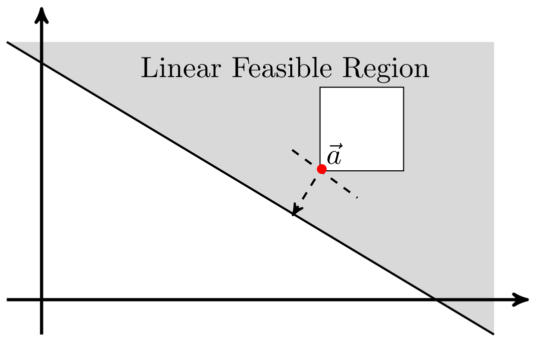
After obtaining a search box which suffices the preset precision requirements, it is necessary to validate this box against the *linear feasible region* represented by the solution set $\text{Sol}(L)$ of all linear constraints. There are three possible cases (see Figure 4a):

Case 1: The search box completely resides outside the linear feasible region. In this case, the box should be discarded without an invocation of the expensive NRA solver.

Case 2: The search box resides partially inside the linear feasible region. Since the interval box constitutes an over-approximation of the (possibly empty) solution set, the real point solutions could either lie inside or outside the feasible region.



(a) Position of a search box and its sample point



(b) Nearest vertex of a search box to hyperplane

Figure 4: Relation between a search box and the linear feasible region

Case 3: The search box completely resides inside the linear feasible region. Distinguishing this case from case 2 is the idea of the following validation procedure.

Let us write $\mathrm{Var}(N) = \{x_1, \ldots, x_n\}$ for all nonlinear and $\mathrm{Var}(L) \setminus \mathrm{Var}(N) = \{y_1, \ldots, y_m\}$ for all pure linear variables that resulted from preprocessing. Let $I_N := I|_{\mathrm{Var}(N)}$ denote the restriction of the given box to the nonlinear variables. Ideally, we would like to decide whether for every point $\vec{x} \in I_N$ there is an assignment for the remaining linear variables $\vec{y} \in \mathbb{R}^m$, such that every linear constraint in L is satisfied, that means

$$\forall \vec{x} \in I_N : \exists \vec{y} \in \mathbb{R}^m : (\vec{x}, \vec{y}) \models L. \tag{1}$$

In case this consistency check fails, we would generate the corresponding *conflict set* Σ_{conf} of all linear constraints violating some point in I_N by

$$\Sigma_{\mathrm{conf}} = \{\sigma \in L \mid \exists \vec{x} \in I_N : \forall \vec{y} \in \mathbb{R}^m : (\vec{x}, \vec{y}) \not\models \sigma\}. \tag{2}$$

This consistency condition can be reformulated as a linear program and decided with the help of a LRA solver. Unfortunately, every different search box I leads to a completely new linear program, wherefore the repeated invocation of the corresponding validation procedure would be reflected in an extraordinary higher runtime. Instead, we propose the following weakening of the *strong consistency condition* (1).

For an arbitrary sample point $\vec{p} \in I_N$ of the nonlinear variables $\{x_1, \ldots, x_n\}$, we query the LRA solver for an assignment of the pure linear variables $\{y_1, \ldots, y_m\}$ that satisfies the already asserted linear constraints (see lines 2–4 in Algorithm 3.4). If this check fails, we practice short-circuiting by taking the infeasible subset of the LRA solver as the conflict set Σ_{conf} (line 11). If the sample point $\vec{p}$ does not conflict any linear constraint, the LRA solver returns a point solution $\vec{b} \in \mathbb{R}^m$ for the pure linear variables (line 5). We either have a case 2 scenario, where the chosen sample point occasionally lies inside the region, or the search box I_N is completely inside such that any chosen point of the box would have been accepted (see Figure 4a). To distinguish these cases, we now test the full containment of the box I_N in the feasible region with the pure linear variables $\vec{y}$ fixed by $\vec{b}$, and accumulate all the linear constraints $\sigma \in L$ intersecting the box as our conflict set Σ_{conf} (lines 6–9). This containment proof is the most difficult part of the validation procedure and discussed below in more detail. Summing up, we check the following *weak consistency condition*

$$\exists \vec{y} = \vec{b}(\vec{p}) \in \mathbb{R}^m : \forall \vec{x} \in I_N : (\vec{x}, \vec{y}) \models L.$$

Since the linear feasible region $\mathrm{Sol}(L)$ is determined by the intersection of the solution sets $\mathrm{Sol}(\sigma)$ for $\sigma \in L$, it suffices to check each linear constraint separately. Rewrite a $\sigma \in L$ as

$$\sigma : \vec{c}^T \vec{x} \sim e + \vec{d}^T \vec{y} \quad \text{with} \quad \vec{c} \in \mathbb{R}^n, \vec{d} \in \mathbb{R}^m, e \in \mathbb{R}, \text{ and } \sim \in \{<, \leq\} \tag{3}$$

to separate the nonlinear variables $\vec{x}$ from the linear variables $\vec{y}$. To check the containment of the whole interval box I_N in $\mathrm{Sol}(\sigma)$, we only need to validate the nearest vertex $\vec{a} \in I_N$ to the boundary $\partial(\mathrm{Sol}(\sigma))$ against σ (see Figure 4b). This nearest vertex corresponds to the point that maximises the left hand side of σ in (3), namely $\vec{a} = \arg\max_{\vec{x} \in I_N} \vec{c}^T \vec{x}$. To obtain this maximum, the $\vec{x}$ variables only need to take their minimum or maximum values in their intervals depending on their coefficients $\vec{c}$. We can calculate it directly via

$$a_i := \begin{cases} \overline{I_{N,i}} & \text{, if } c_i > 0 \\ \underline{I_{N,i}} & \text{, if } c_i \leq 0 \end{cases}.$$

Algorithm 3.4 Validate the given box against the linear feasible region.

```
 1: function VALIDATE(Box $I$)
 2:     $\vec{p} \leftarrow$ SAMPLEPOINT($I|_{\mathrm{Var}(N)}$)
 3:     LRA.ASSERT($\vec{p}$)
 4:     if LRA.CHECK() then
 5:         $\vec{b} \leftarrow \mathrm{LRA.model}|_{\mathrm{Var}(L) \setminus \mathrm{Var}(N)}$
 6:         for all $\sigma \in L$ do
 7:             $\vec{a} \leftarrow$ NEARESTPOINT($\sigma, I|_{\mathrm{Var}(N)}$)
 8:             if $(\vec{a}, \vec{b}) \not\models \sigma$ then
 9:                 $\Sigma_{\mathrm{conf}} \leftarrow \Sigma_{\mathrm{conf}} \cup \{\sigma\}$
10:     else
11:         $\Sigma_{\mathrm{conf}} \leftarrow \mathrm{LRA.infSubset} \setminus \vec{p}$
12:     LRA.REMOVE($\vec{p}$)
13:     return $\Sigma_{\mathrm{conf}}$
```

4 Conclusion

In this paper, we have presented interval constraint propagation as a theory module for the DPLL framework used to efficiently tighten the search space of QFNRA formulas before the invocation of a complete decision procedure. Most practical problems in formal verification contain a large number of linear constraints, and only a small number of nonlinear ones. Hence, the integration of ICP with LRA solvers to avoid the slow convergence problem on linear constraints is useful. Another approach to accelerate the contraction process is to replace the shown interval propagation by the Newton-Raphson method. This variant can ensure a quadratic speed of convergence even for linear constraints, but dedicated methods such as Simplex for solving the linear parts of a constraint stay superior.

References

[1] 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. In *Proceedings of the FMCAD*, pages 81–90. FMCAD Inc, 2010.

[2] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[3] S. Schupp. Interval Constraint Propagation in SMT Compliant Decision Procedures. Master's thesis, RWTH Aachen University, March 2013.