A Generalised Branch-and-Bound Approach and its Application in SAT Modulo Nonlinear Integer Arithmetic

Gereon Kremer, Florian Corzilius, and Erika Ábrahám

RWTH Aachen University, Germany

Abstract. The branch-and-bound framework has already been successfully applied in SAT-modulo-theories (SMT) solvers to check the satisfiability of linear integer arithmetic formulas. In this paper we study how it can be used in SMT solvers for non-linear integer arithmetic on top of two real-algebraic decision procedures: the virtual substitution and the cylindrical algebraic decomposition methods. We implemented this approach in our SMT solver `SMT-RAT` and compared it with the currently best performing SMT solvers for this logic, which are mostly based on bit-blasting. Furthermore, we implemented a combination of our approach with bit-blasting that outperforms the state-of-the-art SMT solvers for most instances.

1 Introduction

Satisfiability checking [7] aims to develop algorithms and tools to check the satisfiability of existentially quantified logical formulas. Driven by the success of SAT solving for propositional logic, fruitful initiatives were started to enrich propositional SAT solving with solver modules for different theories. *SAT-modulo-theories* (*SMT*) *solvers* [6,24] make use of an efficient SAT solver to check the logical (Boolean) structure of formulas, and use different theory solver modules for checking the consistency of theory constraint sets in the underlying theory.

Besides theories like equality logic, uninterpreted functions, bit-vectors and arrays, SMT solvers also support *arithmetic* theories. Apart from interval constraint propagation (ICP), SMT solving for quantifier-free *linear real* arithmetic (*QF_LRA*) often makes use of linear programming techniques like the *simplex* method [15]. For quantifier-free *non-linear real* arithmetic (*QF_NRA*), algebraic decision procedures like the *virtual substitution* (*VS*) method [28] or the *cylindrical algebraic decomposition* method (*CAD*) [12] can be used.

There are several powerful SMT solvers like, *e.g.*, `CVC4` [4], `MathSAT5` [11], `Sateen` [23], `veriT` [9], `Yices2` [17] and `Z3` [26] which offer solutions for quantifier-free *linear integer* arithmetic (*QF_LIA*) problems that mostly build on the ideas in [16]. Closely related to our work is the SMT-adaptation of the *branch-and-bound* (*BB*) framework [27] in `MathSAT5`.

When moving from the real to the integer domain, non-linear arithmetic becomes undecidable. Despite this fact, there are a few SMT solvers that support the satisfiability check of quantifier-free *non-linear integer* arithmetic (*QF_NIA*) formulas, either for restricted domains (in which case the domain becomes finite and the problem becomes decidable) or in an incomplete manner. Some of these SMT solvers apply linearisation [8], whereas other tools such as `iSAT3` [18] and `raSAT` [22] use interval constraint propagation adapted to the integer domain. To the best of our knowledge, all other prominent solvers like `AProVE` [19], `CVC4` or `Z3` apply mainly bit-blasting, which exploits SAT solvers by the use of a binary encoding of bounded integer domains.

Although the satisfiability problem for QF_NIA is undecidable, we see in the employment and adaptation of algebraic decision procedures a promising alternative to achieve incomplete but practically efficient satisfiability checking solutions for QF_NIA problems. Such solutions would be urgently needed in several research areas to open new possibilities and enable novel approaches. A typical example is the field of program verification where, for instance, deductive proof systems often generate QF_NIA formulas as verification conditions (see *e.g.* [3]). Just to mention a second example, for the termination analysis of programs often some QF_NIA termination conditions need to be checked for satisfiability [8]. Currently, non-linear integer arithmetic problems appearing in these areas are often solved using, *e.g.*, theorem proving, linearisation or bit-blasting.

Today's SMT solvers neither exploit adaptations of algebraic QF_NRA decision procedures for finding QF_NIA solutions[1] nor use the BB framework to check QF_NIA formulas for satisfiability. In this paper we investigate these issues. The main contributions of this paper are:

- We show on the example of the CAD method that some algebraic decision procedures for QF_NRA can be adapted to drive their search towards integer solutions. Experimental results show that this approach works surprisingly well on satisfiable problem instances.
- We propose some improvements for the general integration of BB in SMT solving.
- We show on the examples of the VS and the CAD methods, how algebraic QF_NRA decision procedures can be embedded into the BB framework to solve QF_NIA problems.
- Finally, we provide experimental results to illustrate how different decision procedures can be strategically adapted, combined and embedded in the BB framework to tackle the challenging problem of QF_NIA satisfiability checking.

The rest of the paper is structured as follows. We start in Section 2 with some preliminaries on QF_NIA, SMT solving, and the VS and the CAD methods. In Section 3 we present a general framework for BB. Section 4 and 5 are devoted to the integration of the VS and the CAD methods, respectively, into

[1] `Z3` has a command-line option to solve, instead of QF_NIA problems, their QF_NRA relaxations. This way `Z3` can detect unsatisfiability (no real solution is found) or sometimes even satisfiability (the found real solution happens to be integer), but otherwise it returns "unknown".

the BB framework. After an experimental evaluation of the presented approach in Section 7, we conclude the paper in Section 8.

2 Preliminaries

2.1 Quantifier-free non-linear integer arithmetic

Quantifier-free non-linear arithmetic formulas φ (in the following just formulas) are Boolean combinations of *constraints* c which compare *polynomials* p to 0. A polynomial p can be a *constant*, a *variable* x, or a sum, difference or product of polynomials:

$$\begin{array}{rcccccccccccc} p & ::= & 0 & \mid & 1 & \mid & x & \mid & (p+p) & \mid & (p-p) & \mid & (p \cdot p) \\ c & ::= & p=0 & \mid & p<0 & & & & & & & & \\ \varphi & ::= & c & \mid & (\neg\varphi) & \mid & (\varphi \wedge \varphi) & & & & & & \end{array}$$

We use further operators such as disjunction $\vee$, implication $\Rightarrow$ and comparison $>$, $\leq$, $\geq$, $\neq$, which are defined as syntactic sugar the standard way, and standard syntactic simplifications (e.g., we omit parentheses based on the standard operator binding order, write p_1p_2 instead of $p_1 \cdot p_2$, $-p$ instead of $0-p$, etc.).

We use $\mathbb{Z}[x_1,\ldots,x_n]$ to denote the set of all polynomials (with integer coefficients) over the variables $x_1,\ldots,x_n$ for some $n \geq 1$. A polynomial $p \in \mathbb{Z}[x_1,\ldots,x_n]$ is called *univariate* if $n=1$, and *multivariate* otherwise. By $Var(\varphi)$, $Pol(\varphi)$ and $Con(\varphi)$ we refer to the set of all variables, polynomials respectively constraints occurring in the formula φ; especially, $Pol(p \sim 0)$ denotes the polynomial p of a given constraint $p \sim 0$ with $\sim \in \{<,\leq,=,\neq,\geq,>\}$.

Each polynomial $p \in \mathbb{Z}[x_1,\ldots,x_n]$ can be equivalently transformed to the form $a_k x_1^{e_{1,k}} \ldots x_n^{e_{n,k}} + \ldots + a_1 x_1^{e_{1,1}} \ldots x_n^{e_{n,1}} + a_0$ with *coefficients* $a_j \in \mathbb{Z}$ for $0 \leq i \leq n$ and *exponents* $e_{i,j} \in \mathbb{N}_0$ for $1 \leq i \leq n$ and $1 \leq j \leq k$. We call $m_j := x_1^{e_{1,j}} \ldots x_n^{e_{n,j}}$ a *monomial*, $t_j := a_j m_j$ a *term*, and a_0 the *constant part* of p. In the following we assume that polynomials are in the above form with pairwise different monomials; note that this form is unique up to the ordering of the terms. By $p_1 \equiv p_2$ ($\varphi_1 \equiv \varphi_2$) we denote that the polynomials p_1 and p_2 (the formulas φ_1 and φ_2) can be transformed to the same form.

By $\deg(t_j) := \sum_{i=1}^{n} e_{i,j}$ we denote the *degree* of the term t_j. By $\deg(p) := \max_{1\leq j\leq k} \deg(t_j)$ we denote the *degree of* p and by $\deg(x_i,p) := \max_{1\leq j\leq k} e_{i,j}$ the *degree of* x_i *in* p. A polynomial p is *linear*, if $\deg(p) \leq 1$, and *non-linear* otherwise. A formula φ is *linear*, if all polynomials $p \in Pol(\varphi)$ are linear, and *non-linear* otherwise.

We use the standard semantics of arithmetic formulas. In the theory of *quantifier-free non-linear integer arithmetic* (*QF_NIA*), all variables x_i are integer-valued ($Dom(x_i) = \mathbb{Z}$); in *quantifier-free non-linear real arithmetic* (*QF_NRA*) all variables x_i are real-valued ($Dom(x_i) = \mathbb{R}$); in the theory of *non-linear mixed integer-real arithmetic* (*QF_NIRA*), variables can have either domains. We denote by $\varphi_{\mathbb{Z}}$ ($\varphi_{\mathbb{R}}$) that φ is interpreted as a QF_NIA (QF_NRA) formula, and call $\varphi_{\mathbb{R}}$ the *real relaxation* of $\varphi_{\mathbb{Z}}$.

	$<$	$\leq$	$=$	$\geq$	$>$
$a'_0 \in \mathbb{Z}$	$r + a'_0 + 1 \leq 0$	$r + a'_0 \leq 0$	$r + a'_0 = 0$	$r + a'_0 \geq 0$	$r + a'_0 - 1 \geq 0$
$a'_0 \notin \mathbb{Z}$	$r + \lceil a'_0 \rceil \leq 0$	$r + \lceil a'_0 \rceil \leq 0$	`false`	$r + \lfloor a'_0 \rfloor \geq 0$	$r + \lfloor a'_0 \rfloor \geq 0$

Table 1: Simplification of a QF_NIA constraint $(\sum_{i=1}^{k} a_i m_i) + a_0 \sim 0$, where g is the greatest common divisor of $a_1, \ldots, a_k$, $a'_0 := \frac{a_0}{g}$ and $r := \sum_{i=1}^{k} \frac{a_i}{g} m_i$

As a preprocessing step for QF_NIA formulas, we replace inequalities $p \neq 0$ by $p < 0 \vee p > 0$. Furthermore, based on the integrality of all variables, we simplify constraints in the formula according to Table 1. After these simplifications, only the relations $=$, $\geq$ and $\leq$ (but no $<$, $>$ nor $\neq$) appear in the formulas.

The *substitution* of a variable x by a value $v \in \mathit{Dom}(x)$ in a formula φ is denoted by $\varphi[v/x]$. A value $(v_1, \ldots, v_n) \in \mathbb{R}^n$ is a *real root* or *zero* of a polynomial $p \in \mathbb{Z}[x_1, \ldots, x_n]$ if $p(v_1, \ldots, v_n) := p[v_1/x_1] \ldots [v_n/x_n] \equiv 0$. A value $(v_1, \ldots, v_n) \in \mathbb{R}^n$ is a *solution* of a formula φ with $\mathit{Var}(\varphi) = \{x_1, \ldots, x_n\}$ if $\varphi(v_1, \ldots, v_n) := \varphi[v_1/x_1] \ldots [v_n/x_n] \equiv \texttt{true}$.

The *satisfiability checking problem* is the problem to decide whether there exists a solution for a given formula φ. Note that checking the satisfiability of a quantifier-free formula φ with $\mathit{Var}(\varphi) = \{x_1, \ldots, x_n\}$ and checking the validity of the existentially quantified formula $\exists x_1 \ldots \exists x_n . \varphi$ define the same problem.

2.2 SAT-modulo-theories solving

For the satisfiability check of logical formulas over some theories, *SAT-modulo-theories* (*SMT*) *solvers* combine a SAT solver with one or more *theory solvers*. The SAT solver is used for the efficient exploration of the logical structure of the input formula, whereas the theory solver(s), implementing some decision procedures for the underlying theory, are used to check the consistency of sets (conjunctions) of certain theory constraints. The SMT-solving framework is illustrated in Figure 1.

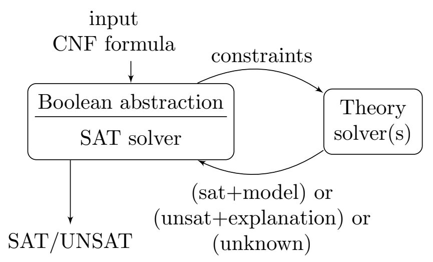


Fig. 1: The SMT solving framework

In this work we consider a less-lazy SMT-solving approach based on DPLL SAT solving. The input formula is brought to negation normal form, negation is applied to atomic constraints, and finally conjunctive normal form (CNF) is built (using Tseitin's transformation), resulting in a conjunction of disjunctions of atomic (not negated) constraints. The SAT solver tries to find a satisfying solution for the *Boolean skeleton* of the input formula, which is the propositional logic formula obtained by replacing each constraint c by a fresh proposition h_c. In other words, the SAT solver tries to determine a set of constraints such that if they have a common solution then the input formula is satisfiable. During its search, the SAT solver works "less lazy", *i.e.*, it asks the theory solver(s) regularly whether the set of those constraints, whose abstraction variables are `true` in the current partial assignment, are together consistent. Note that we do *not* need to pass the negation of constraints with `false` abstraction variables to the theory solver.

The DPLL algorithm, used in most state-of-the-art SAT solvers, executes the following loop: It repeatedly makes a *decision*, assuming a certain value for some proposition. Then it applies *Boolean constraint propagation* (*BCP*), thereby identifying variable assignments that are implied by the last decision. If the propagation succeeds without conflicts, a new decision is made. However, the propagation might also lead to a *conflict*, which means that the current partial assignment cannot be extended to a full satisfying solution. In the latter case *conflict resolution* is applied to determine which decisions led to the conflict. After *backtracking*, *i.e.*, undoing some previous decisions, the SAT solver learns a new clause to exclude the found conflict (and other similar ones) from the future search and continues the search in other parts of the search space.

In the less-lazy SMT-solving context, theory solvers need to be able to work *incrementally*, *i.e.*, to extend a previously received set of constraints with some new constraints and to check the extended set for consistency; similarly, they need to be able to backtrack, *i.e.*, to remove some constraints from their current constraint sets. Furthermore, to enable the SAT solver to resolve theory-rooted conflicts, the theory solver should return an *explanation* if it detects inconsistency, usually in form of an inconsistent subset of its received constraints.

We use our SMT-solving toolbox `SMT-RAT` [14] to implement the approaches described in this work, and to compare the results to other approaches. `SMT-RAT` provides a rich set of SMT-compliant implementations of QF_NRA/ QF_NIA procedures. These procedures are encapsulated in *modules*, which can be composed to an SMT solver according to a user-defined *strategy*. In this work we will use *sequential* strategies only, where a preprocessing module, a SAT solver module and one or more theory solver modules are composed sequentially. The SAT solver sends theory constraints in an incremental fashion to the first theory solver module, which tries to determine whether its received constraints have a common satisfying solution. The first theory solver might also pass on sub-problems[2] for a satisfiability check to the next theory solver module and so on. Each theory solver module returns to its caller module (i) either satisfiability along with a model if requested, (ii) or unsatisfiability and an explanation in form of an infeasible subset of its received formulas, (iii) or unknown. Besides modules for parsing the input problem and transforming it to conjunctive normal form as requested by SAT solving, we use a `MiniSat`-based SAT solver module and theory solver modules implementing the simplex, the VS and the CAD methods, and a theory solver module for bit-blasting.

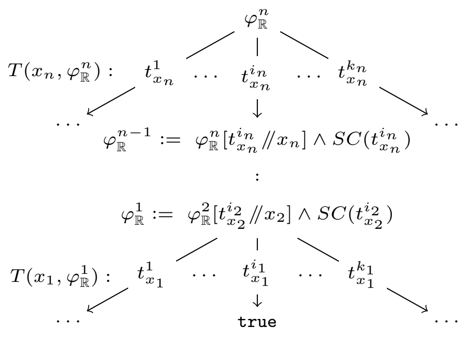
[2] These sub-problems are not necessarily constraints or conjunctions of constraints, but in general formulas with arbitrary Boolean structure.

2.3 Virtual substitution

The *virtual substitution* (*VS*) method [28] is an incomplete decision procedure for non-linear real arithmetic. As we aim at satisfiability checking, we restrict ourselves to the quantifier-free fragment QF_NRA (where we understand all free variables as existentially quantified). Given a QF_NRA formula $\varphi_{\mathbb{R}}$ in variables $Var(\varphi_{\mathbb{R}}) = \{x_1, \ldots, x_n\}$, the VS iteratively eliminates variables that appear at most quadratic in $\varphi_{\mathbb{R}}$. Assume *w.l.o.g.* that we want to eliminate a variable x_n, such that $\deg(x_n, p) \leq 2$ for all $p \in Pol(\varphi_{\mathbb{R}})$.

In the univariate case, *i.e.* $Pol(\varphi_{\mathbb{R}}) \subseteq \mathbb{Z}[x_n]$, we can use the solution equation for quadratic polynomials to determine the real roots of all polynomials in $\varphi_{\mathbb{R}}$. As these real roots separate regions in which each $p \in Pol(\varphi_{\mathbb{R}})$ is sign-invariant, the satisfiability of $\varphi_{\mathbb{R}}$ can be determined by the specification of a set $T(x_n, \varphi_{\mathbb{R}})$ of *test candidates*, being representative elements from sign-invariant regions, and checking whether one of the test candidates satisfies the formula.

Even if the formula contains multivariate polynomials, we can still apply the solution equation to determine possible real roots t_i and the side conditions $SC(t_i)$ for their existence, if we interpret multivariate polynomials $p \in \mathbb{Z}[x_1, \ldots, x_n]$ as univariate polynomials with polynomial coefficients $p \in \mathbb{Z}[x_1, \ldots, x_{n-1}][x_n]$. However, now the results are parametric in $x_1, \ldots, x_{n-1}$, therefore we know neither the existence nor the order of the roots; here, the VS uses $-\infty$ as a test candidate from the left-most sign-invariant region, and infinitesimals ϵ in order to represent with $t + \epsilon$ the sign-invariant region on the right of the root t. As the test candidates might contain fractions, radicals, $-\infty$ and ϵ, instead of the standard substitution $[t_i/x_n]$, a *virtual* substitution $[t_i /\!/ x_n]$ is used to achieve an arithmetic formula $\varphi_{\mathbb{R}}[t_i /\!/ x_n]$ that is satisfiability-equivalent to $\varphi_{\mathbb{R}}[t_i/x_n]$. The rules and their derivation which specify how to construct these satisfiability-equivalent formulas can be found in [28].



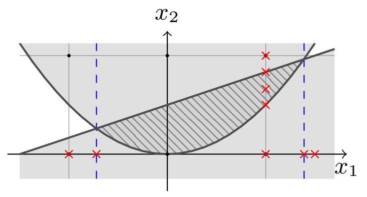
$\varphi_{\mathbb{R}}^n$
$T(x_n, \varphi_{\mathbb{R}}^n):$
$t_{x_n}^1 \;\cdots\; t_{x_n}^{i_n} \;\cdots\; t_{x_n}^{k_n}$
$\varphi_{\mathbb{R}}^{n-1} := \varphi_{\mathbb{R}}^n[t_{x_n}^{i_n} /\!/ x_n] \wedge SC(t_{x_n}^{i_n})$
$\varphi_{\mathbb{R}}^1 := \varphi_{\mathbb{R}}^2[t_{x_2}^{i_2} /\!/ x_2] \wedge SC(t_{x_2}^{i_2})$
$T(x_1, \varphi_{\mathbb{R}}^1):$
$t_{x_1}^1 \;\cdots\; t_{x_1}^{i_1} \;\cdots\; t_{x_1}^{k_1}$
true

Fig. 2: Possible VS depth-first search

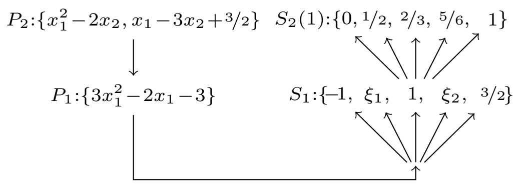
In summary, the VS specifies a finite set $T(x_n, \varphi_{\mathbb{R}})$ of (symbolic) *test candidates* for x_n in $\varphi_{\mathbb{R}}$, and for each test candidate $t \in T(x_n, \varphi_{\mathbb{R}})$ some *side conditions* $SC(t)$, such that

$$\varphi_{\mathbb{R}} \textit{ is satisfiable} \quad \Leftrightarrow \quad \bigvee_{t \in T(x_n, \varphi_{\mathbb{R}})} (\varphi_{\mathbb{R}}[t /\!/ x_n] \wedge SC(t)) \quad \textit{is satisfiable} \,. \qquad (1)$$

In [13] we presented an implementation of the VS for satisfiability checking, which executes a depth-first search for a `true` leaf as illustrated in Figure 2; a solution can be read off the *solution path* from the root to the `true` leaf.



(a) Plot of the solution space and the sample points



P_2:{$x_1^2-2x_2$, $x_1-3x_2+3/2$}
P_1:{$3x_1^2-2x_1-3$}
$S_2(1)$:{0, 1/2, 2/3, 5/6, 1}
S_1:{−1, ξ_1, 1, ξ_2, 3/2}

(b) Schematic overview of the CAD method

Fig. 3: Example of CAD method for $c_1 : x_1^2 - 2x_2 \leq 0$ and $c_2 : x_1 - 3x_2 + 3/2 \geq 0$

2.4 Cylindrical algebraic decomposition

The *cylindrical algebraic decomposition* (*CAD*) is a complete decision procedure for non-linear real arithmetic, which we will use in the SMT solving context again only for checking the satisfiability of QF_NRA formulas. Mathematical definitions of the CAD method can be found in, *e.g.*, [1,2,12]. Due to space restriction, here we give only a high-level description.

Given a formula $\varphi_\mathbb{R}$ in variables $x_1, \ldots, x_n$, the CAD method partitions the search space $\mathbb{R}^n$ into a finite number of disjoint n-dimensional *cells*. Each of the cells is a connected semi-algebraic set over which all polynomials from $Pol(\varphi_\mathbb{R})$ are sign-invariant and thus either all or none of the points in a cell satisfy $\varphi_\mathbb{R}$. The cells are constructed to be *cylindrical*: the projections of any two n-dimensional cells onto the k-dimensional space ($1 \leq k < n$) are either identical or disjoint. The n-dimensional cells with identical k-dimensional projection S form *cylinders* $S \times \mathbb{R}^{n-k}$.

The CAD method uses a two-phase approach to compute such a partitioning. In the *projection* phase, a *projection operator* is applied to the input set $P_n = Pol(\varphi_\mathbb{R})$ of n-dimensional polynomials, yielding a set $P_{n-1} \subset \mathbb{Z}[x_1, \ldots, x_{n-1}]$ of $(n-1)$-dimensional polynomials, whose real roots constitute the boundaries of the $(n-1)$-dimensional projections of the n-dimensional CAD cells. The projection operator is applied recursively until a set $P_1 \subset \mathbb{Z}[x_1]$ of univariate polynomials is obtained. Much work has been done on providing efficient methods for computing preferably small projections, for example in [10,21,25].

In the second phase, the *lifting* or *construction*, the CAD method constructs a *sample point* for each of the cells. It first isolates the real roots $\xi_1, \ldots, \xi_k$ of the polynomials in P_1 which constitute the boundaries of the 1-dimensional projection of the cells which are the intervals $I = \{(-\infty, \xi_1), [\xi_1, \xi_1], (\xi_1, \xi_2), \ldots, (\xi_k, \infty)\}$. We choose a sample point from each of these intervals I_i, resulting in a sample set $S_1 = \{s_1, \ldots, s_{2k+1}\}$. Each sample point $s \in S_1$ from some interval I_i is now *lifted* using the polynomials from P_2: each polynomial from P_2 is partially evaluated on s which results in a set of univariate polynomials whose real roots $\{\xi_1^s, \ldots \xi_{l_s}^s\}$ are again isolated. The real roots $\{\xi_1^s, \ldots \xi_{l_s}^s\}$ specify the boundaries of the 2-dimensional cells in the cylinder $I_i \times \mathbb{R}$ at s: There are l_s surfaces that separate the individual cells within the cylinder $I_i \times \mathbb{R}$, given by $\{(s, \xi_j^s) \mid s \in I_i, 1 \leq j \leq l_s\}$. The lifting procedure is repeated until we obtain n-dimensional sample points that are representatives for the n-dimensional cells that are sign-invariant with respect to the polynomials in P_n. We evaluate $\varphi_{\mathbb{R}}$ for each n-dimensional sample point to check whether one of them satisfies the formula, in which case we obtain a satisfying solution, otherwise the formula is not satisfiable.

The CAD method is illustrated on a 2-dimensional example in Figure 3. Figure 3a depicts the solution space, while Figure 3b visualises the CAD computation. The sample point $(1, {}^{2}\!/_{3})$ satisfies the formula $x_1^2 - 2x \leq 0 \wedge x_1 - 3x_2 + {}^{3}\!/_{2} \geq 0$. Note that we can *choose* a sample point from intervals between real roots. This choice is important when searching for integer solutions: while there is no integer solution for the sample point 1 from the interval (ξ_1, ξ_2), selecting 0 instead of 1 would have resulted in the integer solution $(0, 0)$.

3 A General Branch-and-Bound Framework

A popular approach to check QF_LIA formulas $\varphi_{\mathbb{Z}}$ for satisfiability is the *branch-and-bound* (*BB*) framework [27]. It first considers the relaxed problem $\varphi_{\mathbb{R}}$ in the real domain. If the relaxed problem is unsatisfiable then the integer problem is unsatisfiable, too. Otherwise, if there exists a real solution then it is either integer-valued, in which case $\varphi_{\mathbb{Z}}$ is satisfiable, or it contains a non-integer value $r \in \mathbb{R} \setminus \mathbb{Z}$ for an integer-valued variable x. In the latter case a *branching* takes place: BB reduces the relaxed solution space by excluding all values between $\lfloor r \rfloor = \max\{r' \in \mathbb{Z} \mid r' \leq r\}$ and $\lceil r \rceil = \min\{r' \in \mathbb{Z} \mid r' \geq r\}$ in the x-dimension, described by the formula $\varphi' = \varphi \wedge (x \leq \lfloor r \rfloor \vee x \geq \lceil r \rceil)$. This procedure is applied iteratively, *i.e.*, BB will now search for real-valued solutions for φ'. BB terminates if either an integer solution is found or the relaxation is unsatisfiable. Note that BB is in general incomplete even for the decidable logic QF_LIA.

The most well-known application combines BB with the simplex method. As branching introduces disjunctions and thus in general non-convexity, branching is implemented by case splitting: in one search branch we assume $x \leq \lfloor r \rfloor$, and in a second search branch we assume $x \geq \lceil r \rceil$. Depending on the heuristics, the search can be depth-first (full check of one of the branches, before the other branch is considered), breadth-first (check real relaxations in all current open branches before further branching is applied), or it can follow a mixed strategy.

The combination of BB with the simplex method was also explored in the SMT-solving context [16]. The advantage in this setting is that we have more possibilities to design the branching.

– We can integrate a theory solver module based on the simplex method as described above, implementing BB internally in the theory solver by case splitting. It comes with the advantage that case splitting is always *local* to the current problem of the theory solver and does not affect later problems, and with the disadvantage that we cannot exploit the advantages of *learning*, *i.e.*, to remember reasons of unsatisfiability in certain branches and use this information to speed up the search in other branches.

– Alternatively, given a non-integer solution r for a variable x found by the theory solver on a relaxed problem, we can lift the branching to the SAT solver by extending the current formula with a new clause $(x \leq \lfloor r \rfloor \vee x \geq \lceil r \rceil)$ [5]. As now also the newly added clause must be satisfied in order to satisfy the extended formula, the SAT solver will assign (the Boolean abstraction variable of) either $x \leq \lfloor r \rfloor$ or $x \geq \lceil r \rceil$ to true, *i.e.*, the branching takes place. On the positive side, lifting branching information and branching decisions to the SAT solver allows us to learn from information collected in one branch, and to use this learnt information to speed up the search in other branches. On the negative side, the branching in not local anymore as it is remembered in a learnt clause, therefore it might cause unwanted splittings in later search.

To unify advantages, `MathSAT5` [20] implements a combined approach with theory-internal splitting up to a given depth and splitting at the logical level beyond this threshold.

Following the BB approach in combination with the simplex method, we can transfer the idea also to non-linear integer arithmetic: We can use QF_NRA decision procedures to find solutions for the relaxed problem and branch at non-integer solutions of integer-valued variables. However, there are some important differences. Most notably, the computational effort for checking the satisfiability of non-linear real arithmetic problems is *much* higher than in the linear case. If we have found a real-valued solution and apply branching to find integer solutions, the branching will *refine* the search in the VS and CAD methods: it will create additional test candidates for the VS and new sample points for the CAD method, which will serve as roots for new sub-trees in the search tree. However, the search trees in both branches have a lot in common, that means, a lot of the same work has to be done for both sides of the branches. To prevent the solvers from doing much unnecessary work, we should carefully design the BB procedure.

– Branching should be *lifted* to the SAT solver level to enable *learning*, both in the form of *branching lemmas* as well as *explanations* for unsatisfiability in different branches.
– Learning explanations will allow us to speed up the search by transferring useful information between different branches. However, we need to handle branching lemmas thoughtfully and assure that learnt branching lemmas will not lead to branching for all future sub-problems, but only for "*similar*" ones where the branching will probably be useful.
– As branching refines the search, it should work in an *incremental* fashion without resetting solver states.
– If possible, the search strategies of the underlying QF_NRA decision procedures should be tuned to *prefer integer solutions* (and if they can choose between different integer values, they should choose the most "promising" one).
– Last but not least, as the performance of solving QF_NRA formulas for satisfiability highly improves if different theory solvers implementing different decision procedures are used in combination, a practically relevant BB approach for QF_NIA should support this option.

Algorithm 1 Extended SAT solving algorithm for BB in SMT

```
extended SAT algorithm()
begin
 1 :   while true do
 2 :     if BCP returns no conflict then
 3 :       send newly assigned theory constraints to theory solver
 4 :       check theory consistency
 5 :       if theory solver returned unsat then
 6 :         learn theory conflict
 7 :       end if
 8 :     end if
 9 :     if Boolean or theory conflict occurred then
10 :       try to resolve conflict
11 :       if conflict can be resolved then
12 :         backtrack in SAT and theory solving
13 :       else
14 :         if Line 26 was visited then return unknown
15 :         else return unsat
16 :       end if
17 :     else
18 :       if theory solver returned urgent branching lemmas then
19 :         learn them and branch
20 :       else if not all variables are assigned then
21 :         assign a value to an unassigned variable
22 :       else if theory solver returned unknown then
23 :         if theory solver returned final branching lemmas then
24 :           learn them and branch
25 :         else
26 :           exclude current Boolean assignment from further search
27 :         end if
28 :       else return sat                    // theory solver returned sat
29 :     end if
30 :   end while
end
```

3.1 Processing branching lemmas in DPLL(T) SAT solving

As mentioned before, an SMT solver combines a SAT solver and one or more theory solver modules. First we discuss our general approach of adapting these modules to implement BB for non-linear arithmetic, as described in Algorithm 1.

If we remove the Lines 14, 18-19, 22-27 from Algorithm 1 (printed in italic font) then we achieve the standard SMT framework without BB embedding. This basic algorithm first applies Boolean constraint propagation (BCP, Line 2) to detect implications of current decisions. If BCP does not lead to a conflict, the consistency in the theory domain is checked (Lines 3-4). If the theory constraints, which have to hold according to the current Boolean assignment, are inconsistent (Line 5) then the theory solver provides an explanation (an infeasible subset of its input constraints). We negate this explanation (resulting in a tautology) and add its Boolean abstraction as a new clause to the clause set to exclude this theory conflict from future search (Line 6); If either the BCP led to a Boolean conflict or a theory conflict occurred, then the solver tries to resolve the conflict (Line 10). If the conflict can be resolved (Line 11), then backtracking removes some decisions that led to the conflict; note that also the corresponding theory constraints will be removed from the input constraint list of the underlying theory solver (Line 12). As a result of a successful conflict resolution, a new clause will be learnt that will cause some new implications in the next BCP iteration. Otherwise, if the conflict could not be resolved, the input formula is unsatisfiable (Line 15).

Otherwise, if no conflict occurred, either all variables are assigned, in which case we have found a full solution (Line 28), or we choose one unassigned variable and assign it a certain value (Lines 20-21), which will be propagated when executing the next iteration of the main loop.

We modify this algorithm as follows. Firstly, additionally to `sat` and `unsat`, we allow theory solvers to return also `unknown`. First, this could mean that the underlying theory solving procedure cannot determine the consistency of the set of constraints at hand. Note that, as the logic QF_NIA is undecidable, the only alternative to allow inconclusive answers would be to accept possible non-termination of the theory solver, which does never take place in our implementation. If the Boolean assignment is partial and the theory solver returns `unknown`, the SAT solver continues its search. If a full satisfying Boolean assignment was found by the SAT solver, but the theory solver could not determine whether the solution is consistent in the (integer) theory, then the SAT solver excludes the current Boolean assignment from further search (by learning a clause in Line 26) and continues its search; if the following search detects a satisfying solution then the SMT solver returns `sat`, otherwise it returns `unknown` (Line 14).

The second reason for a theory solver returning `unknown` is that it has found a solution for the real relaxation of its input problem φ, but it is not integer-valued. In this case, the theory solver might return a *branching lemma* of the form

$$(c_1 \wedge \ldots \wedge c_k) \Rightarrow (x \leq \lfloor r \rfloor \vee x \geq \lceil r \rceil) \ , \tag{2}$$

demanding to split the domain of the integer-valued variable x at the non-integer value $r \in \mathbb{R} \setminus \mathbb{Z}$, under the condition that the *branching premise* $c_1 \wedge \ldots \wedge c_k$ with $\{c_1, \ldots, c_k\} \subseteq Con(\varphi)$ holds. Additionally, the theory solver can specify which of the two branches it prefers to start with. We call the Boolean abstraction $(\neg h_{c_1} \vee \ldots \vee \neg h_{c_k} \vee h_{x \leq \lfloor r \rfloor} \vee h_{x \geq \lceil r \rceil})$ of the branching lemma in Eq. 2 a *branching clause* and its last two (possibly fresh) literals *branching literals.*

Branching lemmas can be either *urgent* or *final*[3]. Urgent branching lemmas are immediately abstracted, added to the SAT solver's clause set and used for branching (Lines 18-19). Final branching lemmas are relevant only if the SAT solver has a full satisfying Boolean assignment (Lines 23-24). When a branching clause is added, one of its branching literals (the one that was *not* preferred by the theory solver) will be assigned `false` (thus, if the branching premise

[3] In the experimental results we use final lemmas only.

is `true`, BCP will assign `true` to the preferred branching literal; this way we prevent that both branching literals become `true`, what would result in a theory conflict). Afterwards, we handle the branching clause just as any learnt clause and benefit from the usual reasoning and learning[4] process, which yields the best performance according to our experience.

To prevent unnecessary branchings, we assign to branching literals as decision variables in Line 29 always the value false. Remember that only constraints with `true` abstraction variables will be passed to the theory solver. *I.e.*, only branching clauses whose premise is `true` play a role in the theory, and for those clauses only one of the branching literals.

4 Branch-and-Bound with Virtual Substitution

In this section we present how the VS method as introduced in Section 2.3 can be embedded into the BB framework to check the satisfiability of a given QF_NIA formula $\varphi^n_{\mathbb{Z}}$ over (theory) variables $x_1, \ldots, x_n$. First we apply VS on the real relaxation $\varphi^n_{\mathbb{R}}$ of $\varphi^n_{\mathbb{Z}}$. If we determine unsatisfiability, we know that $\varphi^n_{\mathbb{Z}}$ is also unsatisfiable. Otherwise, if we have found a solution S with the VS for $\varphi^n_{\mathbb{R}}$, as illustrated in Figure 2, then S maps the variables $\mathit{Var}(\varphi^n_{\mathbb{R}}) = \{x_1, \ldots, x_n\}$ to test candidates $S(x_j) = t^{i_j}_{x_j}$ $(1 \leq j \leq n)$. For QF_NIA formulas we can omit to consider strict inequalities as described in Table 1. This saves us from considering test candidates with infinitesimals as introduced in [28] and the comparably higher complexity they entail. Therefore, $S(x_j)$ is either $-\infty$ or of the form $\frac{q_{j,1}+q_{j,2}\sqrt{q_{j,3}}}{q_{j,4}}$ with $q_{j,1}, \ldots, q_{j,4} \in \mathbb{Z}[x_1, \ldots, x_{j-1}]$ (roots parametrised in some polynomials).

If a solution S for the relaxation $\varphi^n_{\mathbb{R}}$ is found then there is a `true` leaf in the search tree, as illustrated in Figure 2. We now try to construct an *integer* solution S^* from the parametrised solution S, as illustrated in Figure 4, traversing the solution path from the `true` leaf backwards. If the test candidate $t^{i_1}_{x_1}$ for x_1 is not $-\infty$, it does not contain any variables, thus we can determine whether its value is integer and set $S^*(x_1)$ to this value. If $t^{i_1}_{x_1} = -\infty$, we can take any integer which is strictly smaller than all the other test candidates in $T(x_1, \varphi^1_{\mathbb{R}})$. Now we iterate backwards: for each test candidate $t^{i_j}_{x_j}$ on the solution path, which is not $-\infty$, we substitute the values $S^*(x_1), \ldots, S^*(x_{j-1})$ for the variables $x_1, \ldots, x_{j-1}$, resulting in $S^*(x_j) := S(x_j)[S^*(x_1)/x_1] \ldots [S^*(x_{j-1})/x_{j-1}]$, which again does not contain any variables and we can evaluate whether its value is integer. If $t^{i_j}_{x_j} = -\infty$ then we evaluate all test candidates from $T(x_j, \varphi^j_{\mathbb{R}})$ whose side conditions hold by substituting $S^*(x_1), \ldots, S^*(x_{j-1})$ for $x_1, \ldots, x_{j-1}$ in the test candidate expressions, and we set $S^*(x_j)$ to an integer value that is strictly smaller than all those test candidate values. We repeat this procedure until either a full integer solution is found or the resulting value in one dimension is not integer.

[4] Note that modern SAT solvers also allow to forget learnt clauses that did not contribute to conflicts recently. This applies also to branching clauses.

$$\begin{tikzcd}
\texttt{true} & t^{i_1}_{x_1} \arrow[l] \arrow[r, dash] \arrow[rr, dashed, bend left, "{[t^{i_1}_{x_1}/x_1]}"] & \varphi^1_{\mathbb{R}} & t^{i_2}_{x_2} \arrow[l] \arrow[r, dash] \arrow[r, dashed, bend left, "{[t^{i_2}_{x_2}/x_2]}"] & \cdots \arrow[r, dashed, bend left, "\cdots"] & t^{i_j}_{x_j} \arrow[l] \arrow[r, dash] & \varphi^j_{\mathbb{R}} & \cdots \arrow[l] & t^{i_n}_{x_n} \arrow[l] \arrow[r, dash] & \varphi^n_{\mathbb{R}} \\
 & \in \mathbb{Z} & & \in \mathbb{Z} & & \notin \mathbb{Z} & & & &
\end{tikzcd}$$

Fig. 4: Solution path from Figure 2 traversed backwards from the leaf to the root

If all test candidate values are integer then VS returns `sat`. Otherwise, if we determine that $S^*(x_j)$ for some j is not integer-valued, then there is some $z \in \mathbb{Z}$ such that $S^*(x_j) \in (z-1, z)$. In this case we return the branching lemma

$$\Big(\bigwedge_{\psi \in \mathit{Orig}_{x_j}(S(x_j))} \psi\Big) \Rightarrow (x_j \leq z-1 \vee x_j \geq z),$$

where $\mathit{Orig}_{x_j}(S(x_j))$ denotes the VS module's received constraints being responsible for the creation of the test candidate $S(x_j)$. We can determine this set recursively with $\mathit{Orig}_{x_j}(S(x_j)) := \mathit{Orig}_{x_j}(c)$ if we used constraint $c \in \mathit{Con}(\varphi^j_{\mathbb{R}})$ for generating the test candidate $S(x_j)$, and where

$$\mathit{Orig}_{x_j}(c) := \begin{cases} c & \text{, if } j = n \\ \mathit{Orig}_{x_{j+1}}(c) & \text{, if } x_j \notin \mathit{Var}(c) \\ \mathit{Orig}_{x_{j+1}}(S(x_{j+1})) \cup \mathit{Orig}_{x_{j+1}}(c') & \text{, if } c' \in \mathit{Con}(\varphi^{j+1}_{\mathbb{R}}) \text{ such that} \\ & \quad c \in \mathit{Con}(c'[S(x_{j+1})/\!/x_{j+1}]) \\ \mathit{Orig}_{x_{j+1}}(S(x_{j+1})) & \text{, otherwise.} \end{cases}$$

Note that the last case occurs if the given constraint is introduced through a test candidate's side condition.

Basically, if we have found a non-integer valued test candidate $S^*(x_j) \notin \mathbb{Z}$, we could still continue the procedure to determine all other non-integer-valued test candidates, but the gain (enabling a heuristics to select on which variable value we want to branch) would come at high computational costs, as we need to compute with nested fractions and square roots. Therefore, we do not consider any other heuristics but branch always on the non-integer value that was found first. In contrast, as we will see in the next section, the CAD methods offers more freedom to design other heuristics.

This procedure is sound, as we do not prune any integer solutions. It is not complete, as it is possible that we branch infinitely often for the same variable at an always increasing or always decreasing value. This procedure can also be used to check a QF_NIRA formula for satisfiability, if we eliminate the real-valued variables first.

5 Branch-and-Bound with the CAD Method

Also the CAD method can be embedded into the BB approach the usual way, however, it offers more flexibility to tune its search towards integer solutions.

Sample point selection. The computation time of the lifting phase heavily depends on the representation size of the numbers involved. The representation of an integer is inherently smaller than that of a fractional number of a similar value due to the lack of a denominator and a smaller numerator. Therefore, when selecting a sample point from a given interval, we always choose an integer value whenever one exists. As a side effect, this is not only faster due to the smaller representation but also generates integer solutions automatically which generally helps to avoid unnecessary branches.

When several integer values are available in a given interval as possible sample points, some of them might lead to a full integer solution whereas some others not. Though there is no generally valid rule to determine more "promising" samples, in practice integers around the middle of the intervals might be a good choice. In our implementation we use this heuristics.

Note that if during lifting an integer extension of the current sample is not possible, we could even stop lifting for the given partial sample, as there is no way to extend it to a full integer solution. However, if at an earlier level we had the choice between different integer values for a cell sample then we cannot conclude unsatisfiability in the current sub-tree. Therefore, even if we cannot choose integer samples, in the current implementation we continue lifting and search for a satisfying real extension of the partial integer sample. If the search leads to a (non-integer) solution, we request branching at the SAT level. Otherwise, if the current branch is unsatisfiable in the real domain, we continue the search in other parts of the search space.

Example 1. Consider $P_2 = \{x_2 + x_1 = 0, x_1 < -1\}$. Projecting x_2 yields $P_1 = \{x_1, x_1 + 1\}$ with real roots $\{-1, 0\}$. To satisfy $x_1 < -1$, we only need to choose a sample for x_1 from $(-\infty, -1)$. Assume we choose -2. Lifting -2 yields the polynomials $\{x_2^2 - 2, -1\}$ with real roots $\{-\sqrt{2}, \sqrt{2}\}$, both being non-integer. All other cells around these roots violate the sign condition. However, we cannot infer unsatisfiability in the integer domain: selecting -4 instead of -2 would have produced the polynomials $\{x_2^2 + 4, -3\}$ with integer roots $\{-2, 2\}$.

Remark 1. We would like to mention an idea, which is not yet implemented but could lead to further improvements. Assume, as discussed above, an integer sample $s = (s_1, \ldots, s_j) \in \mathbb{Z}^j$ for which lifting yields no integer extension. Assume now additionally that in each dimension $i = 1, \ldots, j$ the sample s_i is the only integer point in the respective interval; we say that s is *unique*. In this special case the current sub-tree cannot contain any integer solutions; we can safely stop lifting for s and continue in other parts of the search space.

If we find a solution elsewhere, we can return `sat`. However, if the input formula has no integer solution, CAD needs to return an *explanation* for unsatisfiability. In the real domain, we generate such an explanation by specifying for each full-dimensional sample s (*i.e.*, for each leaf in the lifting tree) the set E_s of all original constraints that are violated by the leaf, and computing a possibly small covering set E which contains at least one constraint from each leaf's set E_s.

Now, if we do not complete lifting for some sub-trees because we could determine unsatisfiability at an earlier level, we cannot use the same approach to generate explanations. Instead, we can proceed as follows: Remember that $s = (s_1, \ldots, s_j) \in \mathbb{Z}^j$ is the sample for which lifting was stopped because s is unique and it has no integer extension. Each s_i samples an interval, whose endpoints are zeros of some polynomials from P_i at $(s_1, \ldots, s_{i-1})$; let $P_i^s \subseteq P_i$ be the set of those polynomials for $i = 1, \ldots, j$ and let $P_i^s = \emptyset$ for $i = j+1, \ldots, n$. Now we follow back the projection tree, and for $i = 1, \ldots, n-1$ we iteratively add to P_{i+1}^s all "projection parents" of all polynomials in P_i^s, *i.e.*, all those polynomials that were used in the projection to generate P_i^s. As a result we achieve a set $P_n^s \subseteq P_n$ of original constraints, which serve as an explanation for the unsatisfiability of the sub-tree rooted at s. We compute this set P_n^s for each unique non-completed sample, build their union, and further extend it with additional constraints from P_n to cover all sets $E_{s'}$ of full-dimensional sample leafs s'. The resulting set is an infeasible subset of the input constraint set.

Remark 2. As the selection of sample points might be crutial for discovering integer solutions, to increase the chances of finding integer solutions, we also experimented with choosing (if possible) multiple sample points for an interval instead of a single one. However, the overhead due to these redundant sample points greatly outweighs any gain, even if only two samples for a single interval are chosen. This is because the redundancy increases with every dimension and often leads to an additional exponential growth.

Sample point lifting order. The order in which sample points are lifted is crucial for fast solution finding. As we want to find integer solutions, we first lift integer sample points before considering non-integer ones. Furthermore, if we already have a partial lifting tree due to a previous incremental call to the CAD method, for further lifting we choose partial integer sample points of high dimension first.

Constructing branching lemmas. If CAD finds a solution $s = (s_1, \ldots, s_n) \in \mathbb{R}^n \backslash \mathbb{Z}^n$, it returns `unknown` and requests branching at the SAT level. We have tried three alternative strategies to generate branching lemmas. The first strategy branches on the value of x_i with $i = \min\{i = 1, \ldots, n \mid s_i \notin \mathbb{Z}\}$. The second strategy is similar but takes the highest index. In both cases, the branching premise is the set of all received constraints; in the future we will also experiment with the set P_n^s (see Remark 1).

The third strategy makes use of the sampling heuristics of the CAD that strongly prefers integers. That means that the longest integer prefix $(s_1, \ldots, s_j) \in \mathbb{Z}^j$ of s cannot be further extended with an integer sample component. This strategy generates the branching lemma[5]

$$C \rightarrow \left(\bigvee_{i=1}^{j} x_i \leq s_i - 1 \vee x_i \geq s_i + 1 \right) . \tag{3}$$

Currently, the branching premise C is again the set of all received constraints. In the future we will also investigate collecting all constraints that reject integer sample points in the vicinity of the first non-integer component s_{j+1}. Let $s_{j+1\downarrow}$

[5] This form of multiple-branch lemmas are handled analogously to the 2-branch-case.

$(s_{j+1\uparrow})$ be s where s_{j+1} is replaced by $\lfloor s_{j+1} \rfloor$ ($\lceil s_{j+1} \rceil$). We define the branching premise by $\{c \in C \mid c(s_{j+1\downarrow}) \equiv \texttt{false} \ \vee \ c(\theta_{j+1\uparrow}) \equiv \texttt{false}\}$.

6 Combination of Procedures

We can often improve the performance for solving QF_NRA formulas if we have different decision procedures at hand and use them in combination, such that theory modules can pass on some sub-problems for satisfiability check to other modules. In `SMT-RAT`, such combinations of decision procedures were already available for QF_NRA problems. In this section we discuss how to extend the approach for QF_NIA and the BB framework, on the examples of theory modules implementing the simplex, VS and CAD methods.

Given a set C of non-linear integer arithmetic constraints, the simplex method can be used to check the consistency of the relaxed linear constraints in C, first neglecting the non-linear ones. If simplex determines that the real relaxation of the linear part of the problem is unsatisfiable, it returns `unsat`. If it finds an integer solution that also satisfies the non-linear constraints, it returns `sat`. If it finds a solution that is not completely integer, but satisfies the real relaxation of the non-linear constraints, it creates a branching lemma and returns `unknown`. Otherwise, it forwards the whole input C to another theory solving module, and passes back the result and, if constructed, the branching lemma to its caller.

For VS, assume that we eliminate the variable x_j ($1 \leq j \leq n$) from the formula $\varphi^j_{\mathbb{R}}$ as illustrated in Figure 2. In general, we can also use the virtual substitution if in some of the polynomials in $\varphi^j_{\mathbb{R}}$ the degree of x_i is higher than 2: We generate all test candidates for x_j from all constraints in which x_j appears at most quadratic. If any of those test candidates leads to a satisfying solution, we can conclude the satisfiability of $\varphi^j_{\mathbb{R}}$. Otherwise, we can pass the sub-problem $\varphi^j_{\mathbb{R}}$ to another theory solving module for a satisfiability check. If it returns `unsat`, we have to consider another path in the search tree of Figure 2. If it returns `sat`, we can use the integer assignment of the variables in the passed sub-problem to construct an integer solution for the remaining variables as explained in Section 4. Finally, if the sub-call returns `unknown` and constructs a branching lemma, also VS returns `unknown` and passes the branching lemma through.

The CAD theory solving module implements a complete decision procedure for QF_NRA, and does for this logic currently not pass on any sub-problems to other solver modules.

7 Experimental Results

We evaluated different sequential strategies $\texttt{M}_1 \rightarrow \ldots \rightarrow \texttt{M}_k$ for solving QF_NIA formulas, using the following modules $\texttt{M}_i$:

- The SAT solver module $\texttt{M}_{\text{SAT}}$ behaves as explained in Section 3.
- $\texttt{M}_{\text{SAT}_{\text{Stop}}}$ works similarly except that it returns `unknown` if an invoked theory solver module returns `unknown`, instead of continuing the search for further

Boolean assignments. The module $\mathtt{M}_{\mathrm{SAT}_{\mathrm{Stop}}}$ provides us a reference: if this module is able to solve a problem then the problem can be considered irrelevant for BB (as BB was not involved).

– The module $\mathtt{M}_{\mathrm{LRA}}$ implements the simplex method with branching lemma generation, es explained in Section 6.
– The theory solver modules $\mathtt{M}_{\mathrm{VS}}$ (implementing VS) and $\mathtt{M}_{\mathrm{CAD}}$ (implementing CAD) check the real relaxation of a QF_NIA input formula. If the relaxation is unsatisfiable they return `unsat`, if they coincidentally find an integer solution they return `sat`, otherwise they return `unknown` (without applying BB).
– The VS module $\mathtt{M}_{\mathrm{VS}_{\mathbb{Z}}}$ constructs branching lemmas as explained in Section 4.
– The CAD modules $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{First}}}$ and $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Last}}}$ construct branching lemmas (Sec. 5) based on the first- resp. last-lifted variable with a non-integer assignment.
– The CAD module $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Path}}}$ constructs branching lemmas which exclude the longest integer prefix of the found non-integer solutions (Eq. 3).
– Bit-blasting is implemented in the module $\mathtt{M}_{\mathrm{IntBlast}}$. In our strategies it will be combined with a preceding incremental variable bound widening module $\mathtt{M}_{\mathrm{IncWidth}}$.

All experiments were carried out on AMD Opteron 6172 processors. Every solver was allowed to use up to 4 GB of memory and 200 seconds of wall clock time.

For our experiments we used the largest benchmark sets for QF_NIA from the last SMT-COMP: APROVE, LEIPZIG (both generated by automated termination analysis) and CALYPTO (generated by sequential equivalence checking). Additionally, we crafted a new benchmark set $\textsc{Calypto}_\infty$ by removing all variable bound constraints from CALYPTO and thereby obtaining unbounded problems (together 8572 problem instances, see headline in Fig. 5e for the size of each set).

Selection of a VS heuristics The `SMT-RAT` strategy $\mathtt{M}_{\mathrm{SAT}_{\mathrm{Stop}}} \rightarrow \mathtt{M}_{\mathrm{VS}}$ could solve 7215 `sat` and 84 `unsat` instances, run out of time or memory for 1146 instances, and returned `unknown` for 127 instances. Applying the `SMT-RAT` strategy $\mathtt{M}_{\mathrm{SAT}} \rightarrow \mathtt{M}_{\mathrm{VS}}$ to those 127 instances, we can solve additional 30 `sat` instances. If we replace the module $\mathtt{M}_{\mathrm{VS}}$ by the $\mathtt{M}_{\mathrm{VS}_{\mathbb{Z}}}$ module, which applies branching lemmas, we can solve further 63 `sat` and 10 `unsat` instances (see Figure 5b).

Selection of a CAD heuristics The `SMT-RAT` strategy $\mathtt{M}_{\mathrm{SAT}_{\mathrm{Stop}}} \rightarrow \mathtt{M}_{\mathrm{CAD}}$ could solve 6656 `sat` and 26 `unsat` instances. The main reason why this approach can already solve more than 77% of the examples lies in the nature of the CAD to choose preferably integer sample points and, of course, in the structure of the benchmark instances. The strategy run out of time or memory for 1835 instances, and returned `unknown` for 55 instances. On these 55 examples, we compared the `SMT-RAT` strategy $\mathtt{M}_{\mathrm{SAT}} \rightarrow \mathtt{M}_{\mathrm{CAD}}$ and 3 other strategies replacing the $\mathtt{M}_{\mathrm{CAD}}$ module by $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{First}}}$, $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Last}}}$ resp. $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Path}}}$. As shown in Table 5c, we find 12 additional `sat` instances with the $\mathtt{M}_{\mathrm{CAD}}$ module. The BB modules $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{First}}}$ and $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Last}}}$ perform very similar and find 13 additional `unsat` instances. This is due to the fact that almost always the assignment of only one variable was not yet integer. With the heuristic in the module $\mathtt{M}_{\mathrm{CAD}_{\mathbb{Z}}^{\mathrm{Path}}}$ we could solve less instances.

$\texttt{RAT}_{\mathbb{Z}}$: $\quad \texttt{M}_{\text{SAT}} \longrightarrow \texttt{M}_{\text{LRA}} \longrightarrow \texttt{M}_{\text{VS}_{\mathbb{Z}}} \longrightarrow \texttt{M}_{\text{CAD}_{\mathbb{Z}}^{\text{First}}}$

$\texttt{RAT}_{\text{blast}}$: $\quad \texttt{M}_{\text{IncWidth}} \longrightarrow \texttt{M}_{\text{IntBlast}}$

$\texttt{RAT}_{\text{blast}.\mathbb{Z}}$: $\quad \texttt{M}_{\text{IncWidth}} \longrightarrow \texttt{M}_{\text{IntBlast}} \longrightarrow \texttt{RAT}_{\mathbb{Z}}$

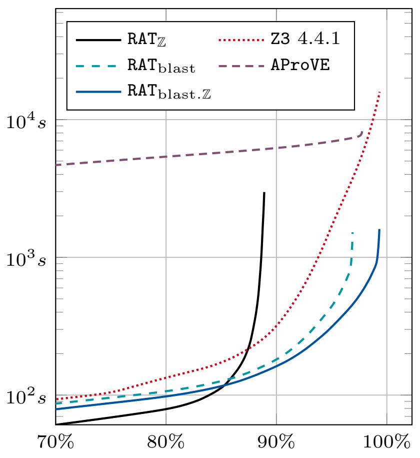
(a) The `SMT-RAT` strategies used for the experimental results

	$\text{VS}_{\mathbb{R}}$ #	$\text{VS}_{\mathbb{R}}$ time	$\text{VS}_{\mathbb{Z}}$ #	$\text{VS}_{\mathbb{Z}}$ time
`sat`	30	714.2	**93**	**487.3**
`unsat`	0	0.0	**10**	**9.2**

(b) Comparison of 2 VS heuristics on 126 (101 `sat`, 25 `unsat`) for BB relevant instances

	$\text{CAD}_{\mathbb{R}}$ #	$\text{CAD}_{\mathbb{R}}$ time	$\text{CAD}_{\mathbb{Z}}^{First}$ #	$\text{CAD}_{\mathbb{Z}}^{First}$ time	$\text{CAD}_{\mathbb{Z}}^{Last}$ #	$\text{CAD}_{\mathbb{Z}}^{Last}$ time	$\text{CAD}_{\mathbb{Z}}^{Path}$ #	$\text{CAD}_{\mathbb{Z}}^{Path}$ time
`sat`	**12**	**137.7**	12	183.5	11	182.7	7	58.4
`unsat`	0	0.0	**13**	**150.8**	13	151.0	2	131.9

(c) Comparison of 4 CAD heuristics on 55 (27 `sat`, 28 `unsat`) for BB relevant instances



(d) Cumulative time to solve instances from all benchmark sets

Benchmark→ Solver↓		APROVE (8129) #	APROVE (8129) time	CALYPTO (138) #	CALYPTO (138) time	LEIPZIG (167) #	LEIPZIG (167) time	CALYPTO_{∞} (138) #	CALYPTO_{∞} (138) time	*all* (8572) #	*all* (8572) time
$\texttt{RAT}_{\mathbb{Z}}$	sat	7283	2294.8	67	71.2	9	260.4	133	298.9	7492	2925.3
	unsat	73	14.3	52	40.7	0	0.0	**3**	$<$ **0.1**	128	55.1
$\texttt{RAT}_{\text{blast}}$	sat	8025	866.3	21	35.6	156	603.3	87	16.0	8289	1521.2
	unsat	12	0.4	5	0.1	0	0.0	0	0.0	17	0.5
$\texttt{RAT}_{\text{blast}.\mathbb{Z}}$	sat	**8025**	**780.7**	**79**	**122.3**	156	511.5	**134**	**21.8**	**8394**	**1436.3**
	unsat	71	42.6	46	127.5	0	0.0	3	0.1	120	170.2
`Z3`	sat	7992	14695.5	78	19.1	158	427.6	126	57.3	8354	15199.5
	unsat	**102**	**595.9**	**57**	**117.6**	0	0.0	3	2.3	**162**	**715.8**
`AProVE`	sat	8025	7052.2	74	559.1	**159**	**696.5**	127	685.2	8385	8993.0
	unsat	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0

(e) Comparison of 3 `SMT-RAT` strategies to currently fastest SMT solvers for QF_NIA

Fig. 5: The column # contains the number of solved instances and the column *time* contains the amount of seconds needed for solving these instances

Combined strategies We crafted three strategies, depicted in Fig. 5a, to combine different theory solver modules[6]. The strategy $\texttt{RAT}_{\text{blast}.\mathbb{Z}}$ combines $\texttt{RAT}_{\text{blast}}$ and $\texttt{RAT}_{\mathbb{Z}}$ by first using bit-blasting up to a width of 4 bits. If this does not yield a solution, it continues to use $\texttt{RAT}_{\mathbb{Z}}$.

We compared these three strategies with the two fastest SMT solvers from the 2015 SMT-COMP for QF_NIA: `Z3` and `AProVE`. Though `CVC4` performed worse than these two solvers, its experimental version solved slightly more instances than `AProVE` in about half of the time; we did not include it here but expect it to perform between `Z3` and `AProVE`. Figure 5e shows that $\texttt{RAT}_{\mathbb{Z}}$ and $\texttt{RAT}_{\text{blast}}$ complement each other well, especially for satisfiable instances. Compared to `Z3` and `AProVE`, $\texttt{RAT}_{\text{blast}.\mathbb{Z}}$ solves more satisfiable instances and does this even faster

[6] Additionally, all of these strategies employ a common preprocessing.

by a factor of more than 10 and 6, respectively. The strategy $\texttt{RAT}_{\mathbb{Z}}$ solves less instances, but, as shown in Figure 5d, this strategy solves the first 85 percent of the examples faster than any other `SMT-RAT` strategy or SMT solver. On unsatisfiable instances, however, `Z3` is still better than `SMT-RAT` while `AProVE` is not able to deduce unsatisfiability due to its pure bit-blasting approach.

We also tested all `SMT-RAT` strategies which use BB, once with and once without using a branching premise. Here we could not detect any notable difference, which we mainly relate to the fact that those problem instances, for which BB comes to application, are almost always pure conjunctions of constraints and involve only a small number of branching lemma liftings. For a more reliable evaluation a larger set of QF_NIA benchmarks would be needed.

8 Conclusion and Future Work

The efficiency of solving quantifier-free non-linear integer arithmetic formulas highly depends on a good strategic combination of different procedures. In this paper we comprised two algebraic procedures, the virtual substitution and the cylindrical algebraic decomposition methods, in a combination with the branch-and-bound approach, which has already been applied effectively in combination with the simplex method. We showed by experimental evaluation that this combination highly complements bit-blasting, the currently most efficient approach for QF_NIA.

The next steps to enhance the strategy for solving QF_NIA formulas could involve interval constraint propagation in order to infer better bounds for the variables. We also plan to further optimise the generation of branching lemmas and of the explanations of unsatisfiability in the theory solving modules.

References

1. Arnon, D.S., Collins, G.E., McCallum, S.: Cylindrical Algebraic Decomposition I: The Basic Algorithm. SIAM Journal on Computing 13(4), 865–877 (1984)
2. Arnon, D.S., Collins, G.E., McCallum, S.: Cylindrical Algebraic Decomposition II: An Adjacency Algorithm for the Plane. SIAM Journal on Computing 13(4), 878–889 (1984)
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Proc. of FMCO'05, chap. Boogie: A Modular Reusable Verifier for Object-Oriented Programs, pp. 364–387. Springer (2006)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proc. of CAV'11. LNCS, vol. 6806, pp. 171–177. Springer (2011)
5. Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Proc. of LPAR'06. LNCS, vol. 4246, pp. 512–526. Springer (2006)
6. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)

7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Borralleras, C., Lucas, S., Navarro-Marset, R., Rodrguez-Carbonell, E., Rubio, A.: Solving Non-linear Polynomial Arithmetic via SAT Modulo Linear Arithmetic. In: Proc. of CADE-22, LNCS, vol. 5663, pp. 294–305. Springer (2009)
9. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Proc. of CADE-22. LNCS, vol. 5663, pp. 151–156. Springer (2009)
10. Brown, C.W.: Improved Projection for Cylindrical Algebraic Decomposition. Journal of Symbolic Computation 32(5), 447 – 465 (2001)
11. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS'13, LNCS, vol. 7795, pp. 93–107. Springer (2013)
12. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Automata Theory and Formal Languages. LNCS, vol. 33, pp. 134–183. Springer (1975)
13. Corzilius, F., Ábrahám, E.: Virtual substitution for SMT solving. In 18th Int. Symp. on Fundamentals of Computation Theory (2011)
14. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In: Proc. of SAT'15. LNCS, vol. 9340, pp. 360–368. Springer (2015)
15. Dantzig, G.B.: Linear programming and extensions. Princeton University Press (1963)
16. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Proc. of CAV'06. LNCS, vol. 4144, pp. 81–94. Springer (2006)
17. Dutertre, B.: Yices 2.2. In: Proc. of CAV'14. LNCS, vol. 8559, pp. 737–744. Springer (2014)
18. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetik constraint systems with complex Boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
19. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Proc. of SAT'07. LNCS, vol. 4501, pp. 340–354. Springer, Lisbon (2007)
20. Griggio, A.: A practical approach to satisfiability modulo linear integer arithmetic. Journal on Satisfiability, Boolean Modeling and Computation 8, 1–27 (2012)
21. Hong, H.: An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In: Proc. of ISSAC'90. pp. 261–264. ACM (1990)
22. Khanh, T.V., Vu, X., Ogawa, M.: raSAT: SMT for Polynomial Inequality. In: Proc. of SMT'14. p. 67 (2014)
23. Kim, H., Somenzi, F., Jin, H.: Efficient term-ITE conversion for satisfiability modulo theories. In: Proc. of SAT'09. LNCS, vol. 5584, pp. 195–208. Springer (2009)
24. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
25. McCallum, S.: An Improved Projection Operation for Cylindrical Algebraic Decomposition of Three-dimensional Space. Journal of Symbolic Computation 5(1), 141 – 161 (1988)
26. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
27. Schrijver, A.: Theory of Linear and Integer Programming. Wiley (1986)
28. Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. 8(2), 85–101 (1997)