Solving Nonlinear Integer Arithmetic with MCSAT[*]

Dejan Jovanović

SRI International

Abstract. We present a new method for solving nonlinear integer arithmetic constraints. The method relies on the MCSat approach to solving nonlinear constraints, while using branch and bound in a conflict-directed manner. We report encouraging experimental results where the new procedure outperforms state-of-the-art SMT solvers based on bit-blasting.

1 Introduction

Integer arithmetic is a natural language to describe problems in many areas of computing. In fields such as operations research, constraint programming, and software verification, integers are the core domain of interest. Automation of reasoning about integers has traditionally focused on linear problems (e.g. [10, 15, 21]). There, powerful methods based on the simplex method and branch-and-bound perform peculiarly well in practice, even though the underlying problem is NP-complete [27]. The branch-and-bound methods use a solver for the reals to find a solution to the problem. If a solution is found in the reals, with a variable x assigned to a non-integer value v, then the solver performs a "split" by introducing a lemma $(x \leq \lfloor v \rfloor) \vee (x \geq \lceil v \rceil)$. Although more powerful methods are available (e.g., cutting planes [14, 20]), branch and bound is still the most prominent method for solving integer problems due to its simplicity and practical effectiveness.

In the case of *nonlinear* integer arithmetic the hurdles stand much higher. The celebrated result of Matiyasevich [25] resolved Hilbert's 10th problem in the negative by showing that satisfiability in the nonlinear case is undecidable. This is in contrast to arithmetic over the reals, where the full theory is decidable [29], and can be solved by effective decision procedures such as virtual term substitution (VTS) [30], cylindrical algebraic decomposition (CAD) [5], and NLSat [19]. In this paper, we focus on the quantifier-free theory of nonlinear integer arithmetic in the usual setting of satisfiability modulo theories (SMT). We are interested in developing a satisfiability procedure that is effective on problems arising in practical applications and can be used in a combination framework to decide problems that involve combinations of theories (e.g., mixed integer-real arithmetic with uninterpreted functions).

[*] The research presented in this paper has been supported by NSF grant 1528153, NASA Cooperative Agreements NNX14AI05A and NNA10DE73C, and by DARPA under agreement number FA8750-16-C-0043.

Most of the current SMT solvers that support nonlinear integer arithmetic (CVC4 [2], Z3 [7], and SMT-RAT [6], APROVE [13]) rely on the *bit-blasting* approach described in [12]. In the bit-blasting approach, an integer satisfiability problem is reduced to a SAT problem by first bounding the integer variables, and then encoding the problem bit-by-bit into a pure SAT problem. The resulting SAT problem can then be discharged with an off-the-shelf SAT solver. Although this approach is limited in its deductive power, it is an effective model finder for practical problems with small solutions. The bit-blasting approach can not detect unsatisfiability unless the problem is bounded. But, since in practical applications many problems are mostly linear, some solvers (e.g. Z3 and CVC4) additionally apply linear and interval reasoning to detect some cases of unsatisfiability. Notably, a recent approach relying on branch-and-bound has been explored in the context of CAD and VTS [22]. Although interesting, the approach provides limited improvements and is only used to supplement the existing techniques.

This paper presents a new method for solving nonlinear integer problems that is based on the MCSat (model-constructing satisfiability) approach to SMT [8, 18], where the application of branch-and-bound has more appeal. The new method reasons directly in the integers, and takes advantage of the MCSat-based solver for nonlinear real arithmetic [19, 17]. Due to the model-constructing nature of MCSat, the new method is able to perform the branch-and-bound "splits" in a *conflict-driven* manner. The conflict-driven branching strategy allows the solver to focus on the relevant areas of the search space and forget the splits that are not useful anymore. This is in contrast to the standard branch-and-bound approaches in SMT where the splitting strategy is delegated to the whims of the underlying SAT solver and is therefore hard to control. The new method can be used in combination with other theories and can also be used to decide mixed real-integer problems where the bit-blasting approach does not apply.

We start by introducing the relevant background and concepts in Section 2. We then present relevant elements of MCSat and the main algorithm in Section 3. We have implemented the new method in the YICES2 [9] solver and we present an empirical evaluation in Section 4 showing that the new method is highly effective.[1]

2 Background

We assume that the reader is familiar with the usual notions and terminology of first-order logic and model theory (for an introduction see e.g. [4]).

As usual, we denote the ring of integers with $\mathbb{Z}$ and the field of real numbers with $\mathbb{R}$. Given a vector of variables $\boldsymbol{x}$ we denote the set of polynomials with integer coefficients and variables $\boldsymbol{x}$ as $\mathbb{Z}[\boldsymbol{x}]$. A polynomial $f \in \mathbb{Z}[\boldsymbol{y}, x]$ is of the

[1] YICES2 won the nonlinear categories of the 2016 SMT competition `http://smtcomp.sourceforge.net/2016/`.

form

$$f(\boldsymbol{y}, x) = a_m \cdot x^{d_m} + a_{m-1} \cdot x^{d_{m-1}} + \cdots + a_1 \cdot x^{d_1} + a_0 \ ,$$

where $0 < d_1 < \cdots < d_m$, and the coefficients a_i are polynomials in $\mathbb{Z}[\boldsymbol{y}]$ with $a_m \neq 0$. We call x the *top variable* and the highest power d_m is the *degree* of the polynomial f. We denote the set of variables appearing in a polynomial f as $\mathsf{vars}(f)$ and call the polynomial *univariate* if $\mathsf{vars}(f) = \{x\}$ for some variable x. Otherwise the polynomial is *multivariate*, or a constant polynomial (if it contains no variables). A number $\alpha \in \mathbb{R}$ is a *root of the polynomial* $f \in \mathbb{Z}[x]$ iff $f(\alpha) = 0$. We denote the set of real roots of a univariate polynomial f as $\mathsf{roots}(f)$.

A *polynomial constraint* C is a constraint of the form $f \nabla 0$ where f is a polynomial and $\nabla \in \{<, \leq, =, \neq, \geq, >\}$. If a constraint C is over a univariate polynomial $f(x)$ we also call it univariate. The solution set of a univariate constraint C in $\mathbb{R}$ is a set of intervals with endpoints in $\mathsf{roots}(f) \cup \{-\infty, \infty\}$ that we denote by $\mathsf{feasible}(C)$. Given a set of univariate constraints $\mathcal{C} = \{C_1, \ldots, C_n\}$, we denote the solution set of $\mathcal{C}$ by $\mathsf{feasible}(\mathcal{C}) = \cap_i \mathsf{feasible}(C_i)$.

An atom is either a polynomial constraint or a Boolean variable, and formulas are defined inductively with the usual Boolean connectives ($\wedge$, $\vee$, $\neg$). Given a formula $F(\boldsymbol{x})$ we say that a type-consistent variable assignment $\boldsymbol{x} \mapsto \boldsymbol{a}$ satisfies F if the formula F evaluates to $\top$ in the standard semantics of Booleans and integers. If there is such a variable assignment, we say that $\mathcal{F}$ is *satisfiable*, otherwise it is *unsatisfiable*.

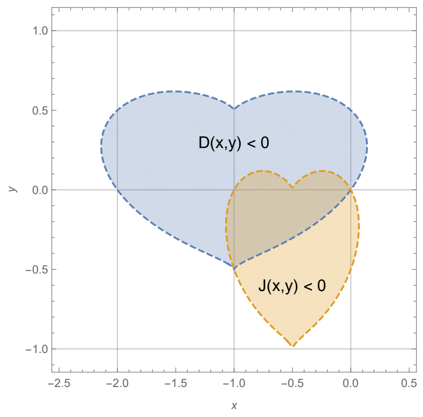

Fig. 1. Solution space of constraints from Example 1.

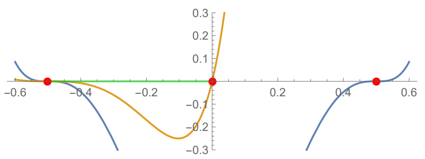

Fig. 2. Roots of $D(-1, y)$ and $J(-1, y)$ from Example 1, with real solutions of $(D < 0) \wedge (J < 0)$ marked in green.

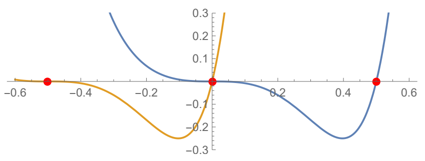

Fig. 3. Roots of $D(0, y)$ and $J(0, y)$ from Example 1, with no solution for $(D < 0) \wedge (J < 0)$.

Example 1. Consider the "heart" polynomial $H(x,y) = (x^2+y^2-1)^3 - x^2y^3$, and its two instances $D(x,y) = H(x+1,2y)$, and $J(x,y) = H(2x+1,2y-1)$. From D and J we can construct two polynomial constraints

$$D(x,y) < 0 \ , \qquad\qquad J(x,y) < 0 \ . \tag{1}$$

The solution space of (1) in $\mathbb{R}^2$ is presented in Figure 1. Considering the real geometry of Figure 1, integer solutions might only exist for $x = -1$ and $x = 0$. Substituting the values -1 and 0 in the constraints gives univariate polynomial constraints

$$\mathcal{C} = \{\ D(-1,y) < 0,\ J(-1,y) < 0\ \}\ , \quad \mathcal{D} = \{\ D(0,y) < 0,\ J(0,y) < 0\ \}\ .$$

With symbolic root isolation[2] for univariate polynomials, we can compute the real solutions of these sets of constraints as

$$\begin{aligned}\mathsf{feasible}(\mathcal{C}) &= (-0.5,0) \cap (-0.5,0.5) = (-0.5,0)\ ,\\ \mathsf{feasible}(\mathcal{D}) &= (-0.5,0) \cap (0,0.5) = \emptyset\ .\end{aligned}$$

Depiction of the roots and feasible intervals of $\mathcal{C}$ and $\mathcal{D}$ is presented in Figures 2 and 3, respectively.

Since in both cases the solution sets do not include any integer points, we can conclude that (1) does not have integer solutions and is therefore unsatisfiable.

3 Algorithm

As Example 1 illustrates, one way to solve integer problems is to understand the real geometry of the underlying problem, and use it to enumerate the potential integer solutions. The new method we present here explores the real geometry through the NLSat solver within the MCSat approach to SMT. We start by introducing the relevant parts of the MCSat framework. The purpose of the MCSat exposition is to make the paper self-contained and to simplify and generalize the concepts already presented in [8, 18]. A reader familiar with MCSat can skip directly to the integer-specific Section 3.4.

The MCSat architecture consists of the core solver that manages the relevant terms, the solver trail, and the reasoning plugins. The core solver drives the solving process, and is responsible for dispatching notifications and handling requests from the plugins, while the plugins reason about the content of the trail with respect to the set of currently relevant terms. In the case of purely integer problems, the only relevant plugins are the arithmetic plugin and the Boolean plugin. The most important duty of the core is to perform conflict analysis when the reasoning plugins detect a conflict state. In order to check satisfiability of a formula F, the core solver notifies active plugins about the formula. The plugins analyze the formula and report all the relevant terms of F back to the core.

[2] Root isolation for univariate polynomials with integer coefficients can be done efficiently with algorithms based on Strum sequences or Descartes' rule of signs [1].

Once the relevant terms are collected, the core adds the assertion $(F \rightsquigarrow \top)$ to the trail, and the search starts as described in Section 3.5.

In this paper we focus on the details of the base calculus and the needed integer reasoning capabilities. For more information on other plugins and combination of theories, we refer the interested reader to [18].

3.1 The Trail

The central data structure in the MCSat framework is the solver trail. It is a generalized version of the trail found in modern SAT solvers and is a representation of the partial (and potentially inconsistent) model being constructed by the solver. The purpose of the trail is to maintain the assignment of relevant terms so that, if the satisfiability algorithm terminates with a positive answer, the satisfying assignment can simply be read off the trail.

Relevant terms are variables and Boolean terms, excluding negation. Intuitively, when invoking theory-specific semantics to evaluate a compound Boolean term t or its negation $\neg t$, the relevant terms are the term t itself and the closest sub-terms of t that are needed to compute its value.

Example 2. Consider the terms $t_1 \equiv (x + y^2 < z)$ and $t_2 \equiv (\neg b_1 \wedge (b_2 \vee b_3))$. In order to compute the value of t_1 or $\neg t_1$ under integer semantics, we need to know the values of terms x, y, and z. For evaluation of t_2 or $\neg t_2$, we need to know the values of terms b_1 and $(b_2 \vee b_3)$. The set of relevant terms therefore must include the terms $\{x, y, z, (x+y^2 < z), b_1, b_2, b_3, (b_2 \vee b_3), (\neg b_1 \wedge (b_2 \vee b_3))\}$.

A *trail* is a sequence of trail elements, where each element is either a *decision*, or a *propagation*. A decision is an assignment of a value v to a relevant term t, denoted as $t \mapsto v$. A propagation is an implied assignment of a value v to a relevant term t, and E is an explanation of the implication, denoted as $t \overset{E}{\rightsquigarrow} v$. In both cases, we say that t *is assigned* in M.[3] In order to properly define an explanation, we first explain how the trail can be used to evaluate terms.

A trail can be seen as a partial model under construction and we can use the trail to evaluate compound terms based on the values of their sub-terms. A term t (and $\neg t$, if Boolean) *can be evaluated* in the trail M if t itself is assigned in M, or if all closest relevant sub-terms of t needed for evaluation are assigned in M (and its value can therefore be computed). Note that some terms can be evaluated in two different ways (by direct assignment, or by evaluation of sub-terms), potentially resulting in two different values. In order to account for this ambiguity, we define an evaluation predicate $\mathsf{evaluates}[M](t, v)$ that returns **true** if the term t can evaluate to the value v in trace M. In addition, we define $\mathsf{reasons}[M](t, v)$ to be the set of relevant sub-terms of t that were used in evaluating t to v.

Definition 1 (Evaluation Consistency). *A trail M is* evaluation-consistent *if there is no term that evaluates to two different values in M.*

[3] For simplicity, we denote formulas asserted to the solver as propagations with no explanation.

Example 3. In the trail $M = [\![\, (x < 0) \mapsto \top, x \mapsto 0 \,]\!]$ the following hold.

– Terms $(x < 0)$ and x are assigned and can be evaluated to $\top$ and 0, respectively, with $\mathsf{reasons}[M](x < 0, \top) = \{x < 0\}$ and $\mathsf{reasons}[M](x, 0) = \{x\}$.
– Term $(x < 0)$ can also be evaluated to $\bot$, since its closest relevant sub-term x is assigned, with $\mathsf{reasons}[M](x < 0, \bot) = \{x\}$.
– The term $(x^2 = 0)$ can be evaluated to $\top$ since the only closest relevant sub-term x is assigned, with $\mathsf{reasons}[M](x^2 = 0, \top) = \{x\}$.
– The term $(x^2 = 0) \vee \neg(x < 0)$ *can not* be evaluated, since its closest relevant sub-terms are $(x^2 = 0)$ and $(x < 0)$, but $(x^2 = 0)$ is not assigned.
– The trail M is *not evaluation consistent* because the term $(x < 0)$ can evaluate to two different values.

The following lemma shows that evaluation consistency is a fundamental property of a trail in model-constructing satisfiability methods.

Lemma 1. *Given a formula F, if an evaluation-consistent trail M assigns all relevant sub-terms of F, and* $\mathsf{evaluates}[M](F, \top)$*, then the model induced by M satsifies F.*

In the MCSat setting, we require the reasoning plugins to ensure evaluation-consistency for the terms they are responsible for. Usually, a plugin is responsible for terms relevant to a theory, i.e., constraints of the theory and values for terms of the theory's principle types. The plugin must detect evaluation inconsistencies in the trail for the constraints they are responsible for, and not introduce any new ones when making assignment decisions.

Explanations. We now define the explanations appearing in the trail propagations. An *explanation E* describes a valid substitution of a term with another term, under given assumptions, and is of the form

$$[A_1, \ldots, A_n] \implies \{t \mapsto s\} \ .$$

A propagation is only allowed to appear in the trail if the accompanying explanation is valid. In a trail $[\![\; M, \; t \overset{E}{\rightsquigarrow} v, \ldots \;]\!]$, the explanation E is a *valid explanation* for the marked propagation if

1. the formula $A_1 \wedge \ldots \wedge A_n \implies t = s$ is valid;
2. each assumption A_i can evaluate to $\top$ in M; and
3. the substitution term s can evaluate to v in M.

The three examples below illustrate the three typical propagation types.

Example 4 (Boolean Propagation). Consider the trail

$$[\![\; (x \vee y \vee z) \rightsquigarrow \top, \;\; x \mapsto \bot, \;\; y \mapsto \bot \;]\!] \ .$$

Boolean propagation over clauses is the core deduction mechanism in a SAT solver. When all but one literals of a clause are assigned to $\bot$ in the trail, we can propagate that the unassigned literal must have the value $\top$. In the example above, we can propagate $z \overset{E}{\rightsquigarrow} \top$, where the explanation E is

$$[(x \lor y \lor z), \neg x, \neg y] \implies \{z \mapsto \top\} \enspace .$$

Example 5 (Evaluation Propagation). Consider the trail $[\![\ x \mapsto 0, y \mapsto 1\ ]\!]$ and the atom $(x + y < 0)$. Since all the variables of the atom are assigned in the trail, we can propagate its value as $(x + y < 0) \overset{E}{\rightsquigarrow} \bot$. The explanation E of this propagation can be $[\neg(x + y < 0)] \implies \{(x + y < 0) \mapsto \bot\}$.

Example 6 (Value Propagation). Consider the trail

$$[\![\ (x \leq z) \rightsquigarrow \top,\ \ (z \leq y) \rightsquigarrow \top,\ \ x \mapsto 0,\ \ y \mapsto 0\ ]\!] \enspace .$$

Since $0 \leq z \leq 0$, we can propagate $z \overset{E}{\rightsquigarrow} 0$. In order to explain this propagation we can use the following valid explanation E

$$[x \leq z, z \leq y, x = y] \implies \{z \mapsto x\} \enspace .$$

Note that the explanation E introduces a new term in the set of assumptions.

3.2 Unit Reasoning

As Lemma 1 states, by maintaining evaluation consistency, we can guarantee that the model induced by the trail is a consistent assignment. In order to make the checking of evaluation consistency more operational, we use evaluation watch lists and the concept of unit consistency.

Let t_0 be a relevant compound term and its closest relevant sub-terms be $t_1, \ldots, t_n$. We call $w = (\mathsf{eval}\ t_0\ t_1\ \ldots\ t_n)$ an *evaluation watch-list.* Given a trail M, if all but one elements of w are assigned in M, with t_i being the one unassigned element, we say that w has become *unit* in M with respect to t_i. For a given a set of relevant terms $\mathcal{R}$, and a term $t \in \mathcal{R}$, we denote by $\mathsf{units}(R, t)$ the set of evaluation watch lists for the terms in $\mathcal{R}$ that are unit in M with respect to the term t.

Example 7 (Boolean unit constraints). Consider the clause $(x \lor y)$. This clause, with its sub-terms x and y, corresponds to the evaluation watch list

$$w = (\mathsf{eval}\ (x \lor y)\ x\ y)) \enspace .$$

Now consider the trails

$$M_1 = [\![\ (x \lor y) \rightsquigarrow \top,\ \ x \mapsto \bot\ ]\!] \enspace , \qquad M_2 = [\![\ x \mapsto \bot,\ \ y \mapsto \bot\ ]\!] \enspace .$$

– In trail M_1, variable y is unassigned making w unit with respect to y. Through Boolean reasoning, we can deduce that variable y has to be assigned to $\top$.

– In trail M_2, on the other hand, w is unit with respect to term $(x \vee y)$. Through Boolean reasoning, we can deduce that term $(x \vee y)$ must be assigned to $\bot$.

Example 8 (Arithmetic unit constraints). Consider the constraint $(x \leq y^2)$. This constraint with its relevant sub-terms x and y corresponds to the evaluation watch list

$$w = (\textsf{eval}\ (x \leq y^2)\ x\ y)\ .$$

Now consider the trails

$$M_1 = [\![\ (x \leq y^2) \rightsquigarrow \bot,\ \ x \mapsto 2\ ]\!]\ , \qquad\qquad M_2 = [\![\ x \mapsto 2,\ \ y \mapsto 2\ ]\!]\ .$$

– In trail M_1, variable y is unassigned making w unit with respect to y. Through arithmetic reasoning, we can deduce that y can only be assigned to a value in $\left(-\sqrt{2}, \sqrt{2}\right)$.
– In trail M_2, on the other hand, w is unit with respect to $(x \leq y^2)$. Through arithmetic reasoning, we can deduce that the term $(x \leq y^2)$ must be assigned to $\top$.

Evaluation watch lists are enough to capture clauses that become unit during Boolean constraint propagation (BCP) process in modern SAT solvers. But, as the examples above show, they can be used to reason about arbitrary Boolean and non-Boolean terms.

Definition 2 (Unit Consistency). *Assume a set of relevant terms $\mathcal{R}$ and a trail M. We call $\mathcal{R}$* unit consistent *in M if, for each term $t \in \mathcal{R}$ that is not yet assigned in M, there exists a value v such that the assignment $t \mapsto v$ is evaluation-consistent with M.*

In a unit inconsistent trail, there is a term that can not be correctly assigned to a value. This is a sign that the trail is in conflict and we need to revise it. On the other hand, in a unit consistent trail, for every unassigned term there is a value that the term can be assigned to without breaking evaluation consistency.

From an implementation standpoint, evaluation watch lists provide an efficient mechanism for detecting unit inconsistencies due to the following.

1. Unit watch lists can be detected efficiently by watching 2 terms in every evaluation watch lists (generalization of the two-watched-literals approach from [26]).
2. Once a watch-list becomes unit, reasoning about the underlying constraint is simplified by the fact that the constraint involves only one variable (e.g., BCP on unit clauses, or finding solutions of univariate arithmetic constraints).

3.3 Conflict Analysis

We call a clause $C \equiv (L_1 \vee \ldots \vee L_n)$ a *conflict clause* in a trail M, if each literal L_i can be evaluated to $\bot$ in M. If the trail M is unit-inconsistent or evaluation-inconsistent, it is the responsibility of the reasoning plugin that detected the inconsistency to produce a *valid* conflict clause.

Example 9 (Boolean conflicts). Consider the trail

$$M = [\![\ (x \wedge y) \mapsto \top,\ \ y \overset{E}{\rightsquigarrow} \bot\ ]\!]\ .$$

Trail M is not evaluation consistent since the term $(x \wedge y)$ can evaluate to both $\top$ and $\bot$ is M. The Boolean reasoning plugin can respond to this inconsistency by returning a clause $(\neg(x \wedge y) \vee y)$. This clause is a conflict clause since all literals evaluate to $\bot$ in M, and is also a valid statement of Boolean logic.

Example 10 (Arithmetic Conflicts). Consider the set of relevant terms $\mathcal{R} = \{x, y, (y < z), (z < x)\}$ and the trail

$$M = [\![\ (y < z) \mapsto \top,\ \ (z < x) \mapsto \top,\ \ y \mapsto 0,\ \ x \mapsto 0\ ]\!]\ .$$

Using the watch list mechanism, the arithmetic reasoning plugin can detect that we have two unit constraints with respect to z. These unit constraints on z imply that $z > 0$ and $z < 0$, making the trail M unit inconsistent. The plugin can respond to this inconsistency by reporting a conflict clause

$$(y < z) \wedge (z < x) \Rightarrow (y < z) \equiv \neg(y < z) \vee \neg(z < x) \vee (y < z)\ .$$

This clause is a conflict clause since all of its literals evaluate to $\bot$ in M, and it is also a valid statement of arithmetic.

Explanations of Propagations. During the conflict-analysis process, we resolve any propagated terms out of the conflict clause. We do so by using the substitution provided by the explanation. Since the conflicts are always represented by single clauses, the substitutions appearing in explanations will only be applied to clauses. Moreover, they will not indiscriminately substitute all occurrences of the target term. Instead, they will rely on a given trail M to select which occurrences to replace. We denote this *evaluation-based substitution* of term t with term s in a clause C as $C\{t \mapsto s\}_M$, and define it as follows.

$$(L_1 \vee \ldots \vee L_n)\{t \mapsto s\}_M = (L_1\{t \mapsto s\}_M \vee \ldots \vee L_n\{t \mapsto s\}_M)$$
$$L\{t \mapsto s\}_M = \begin{cases} L\{t \mapsto s\} & \text{if } s \in \mathsf{reasons}[M](L, \bot)\ , \\ L & \text{otherwise}\ . \end{cases}$$

Intuitively, the substitution of t by s in the conflict clause C should only replace the occurrences of t that were needed to evaluate the clause to $\bot$.

Using substitution as the basis of our explanations allows us to fully extend the mechanics of a modern SAT solver to first-order reasoning. In Boolean satisfiability, the main rule of inference is Boolean resolution. Since we are working with more general theories, we generalize the resolution to a variant of ground paramodulation [28] we call *conflict-directed paramodulation.* The paramodulation rule relies on substitution as a core operation. Both resolution and paramodulation rules are shown side-by-side in Figure 4. We denote by $\mathsf{resolve}[M](E, C)$

$$\frac{A \vee x \qquad \neg x \vee B}{A \vee B} \qquad\qquad \frac{A \vee (t = s) \qquad B}{A \vee B\{t \mapsto s\}}$$

Fig. 4. Boolean resolution and ground paramodulation rules. The Boolean rule can be seen as an instance of ground paramodulation with the substitution $\{x \mapsto \top\}$.

the result of applying the paramodulation rule, with evaluation-based substitution, to the explanation E and clause C. More precisely, if $E = [A_1, \ldots, A_n] \implies \{s \mapsto t\}$ then

$$\mathsf{resolve}[M](E, C) = (\neg A_1 \vee \ldots \vee \neg A_n \vee C\{t \mapsto s\}_M) \enspace .$$

Example 11 (Resolution). Consider the trail

$$M = [\![\, (y = x + 1) \rightsquigarrow \top,\ \ (y \le z) \rightsquigarrow \top,\ \ (z \le x) \rightsquigarrow \top,\ \ x \mapsto 0,\ \ y \overset{E}{\rightsquigarrow} 1 \,]\!] \enspace .$$

The first three entries in the trail are assertions that we are checking for satisfiability, followed by a decision that assigns x to 0. As x is assigned to 0, the constraint $y = x + 1$ becomes unit in y and implies that $y = 1$. The explanation of the propagation is

$$E : [y = x + 1] \implies \{y \mapsto x + 1\} \enspace .$$

The constraints $(y \le z)$ and $(z \le x)$ are unit in variable z and assigned to $\top$. These two constraint simplify to $1 \le z \le 0$ in M and therefore imply an inconsistency on variable z. This inconsistency can be explained by a conflict clause

$$(y \le z) \wedge (z \le x) \Rightarrow (y \le x) \equiv \neg(y \le z) \vee \neg(z \le x) \vee (y \le x) \equiv C_0 \enspace .$$

The clause C_0 is a valid starting point for conflict analysis because it is a valid statement and can be evaluated to $\bot$ in M.

In order to resolve the conflict, we can go back in the trail and resolve the trail elements from the conflict clause one by one. The top propagation to resolve is $y \overset{E}{\rightsquigarrow} 1$, and we can use conflict-directed paramodulation to obtain

$$\begin{aligned} C_1 \equiv \mathsf{resolve}[M](E, C_0) &\equiv \neg(y = x + 1) \vee C_0\{y \mapsto x + 1\}_M \\ &\equiv \neg(y = x + 1) \vee \neg(y \le z) \vee \neg(z \le x) \vee (x + 1 \le x) \\ &\equiv \neg(y = x + 1) \vee \neg(y \le z) \vee \neg(z \le x) \enspace . \end{aligned}$$

Note that we apply the substitution only to the last literal in C_0 since that is the only literal where y was used to evaluate it to $\bot$, i.e. $y \in \mathsf{reasons}[M](y \le x, \bot)$. The new clause C_1 is still in conflict (all literals evaluate to $\bot$), and we can resolve the remaining literals as usual to obtain an empty clause and conclude that the assertions are unsatisfiable.

It is worth noting that the MCSat presentation from [8, 18] did *not allow* propagation of non-Boolean values. This was precisely because the underlying resolution process was based on Boolean resolution.

3.4 Integer Reasoning

The nonlinear integer reasoning we describe bellow assumes an existing MCSat plugin for reasoning about nonlinear real arithmetic. We use the existing NLSat plugin [19, 17] to provide this functionality and, in order to extend it to integer reasoning, we extend the plugin to detect unit inconsistencies over integer constraints, and explain those inconsistencies with appropriate conflict clauses.

Given a trail M and a variable y, the procedure $\mathsf{explain}_{\mathbb{R}}(M, y)$ explains the unit inconsistencies in the reals, and the procedure $\mathsf{explain}_{\mathbb{Z}}(M, y)$ explains the unit inconsistencies in the integers. Both procedures return a valid conflict clause, where the $\mathsf{explain}_{\mathbb{R}}(M, y)$ procedure is inherited from NLSat.

For each arithmetic variable y, we maintain the set of nonlinear constraints $\mathcal{C} = \{C_1, \ldots, C_n\}$ that are unit with respect to y. This means that these constraints are asserted in the trail, and that all variables other than y from $\mathcal{C}$ are also assigned in the trail. We can therefore simplify each constraint in $\mathcal{C}$ by substituting the values of assigned variables and obtain a set of nonlinear constraints $\mathcal{C}_u$ that is *univariate* in y. From there, by isolating the roots of the polynomials involved, we can compute the set $\mathsf{feasible}(\mathcal{C}_u)$ of values that y can take in the context of the current trail. If $\mathsf{feasible}(\mathcal{C}_u)$ contains an integer point, then the trail is feasible with respect to the variable y, otherwise the trail is infeasible and we need to report a conflict.

If $\mathsf{feasible}(\mathcal{C}_u)$ is an empty set, then there is no possible value for y and we are in a *conflict over* $\mathbb{R}$, so we can employ $\mathsf{explain}_{\mathbb{R}}(M, y)$ to explain the conflict and produce a valid conflict clause. Otherwise, the solution set for y is a set of intervals

$$\mathsf{feasible}(\mathcal{C}_u) = (l_1, u_1) \cup (l_2, u_2) \cup \cdots \cup (l_n, u_n) \ ,$$

with real endpoints, where no interval contains an integer point. Here, again, we will reduce conflict explanation to $\mathsf{explain}_{\mathbb{R}}$, but by explaining several conflicts. Consider the (non-integer) point $m = (l_1 + u_1)/2$ and the following two trails

$$M_1 = [\![\ M,\ \ (y \leq \lfloor m \rfloor) \mapsto \top\ ]\!] \ , \qquad M_2 = [\![\ M,\ \ (y \geq \lceil m \rceil) \mapsto \top\ ]\!] \ .$$

These two trails are the branches of M around the point m.

The trail M_1 is unit inconsistent in $\mathbb{R}$ with respect to y and we can therefore apply $\mathsf{explain}_{\mathbb{R}}$ to obtain a conflict clause. Since the inconsistency was initiated by adding the constraint $(y \leq \lfloor m \rfloor)$, the conflict clause must include this constraint and is therefore of the form

$$\mathsf{explain}_{\mathbb{R}}(M_1, y) \equiv \neg(y \leq \lfloor m \rfloor) \vee D_1 \ .$$

Moreover, we know that all literals in the clause D_1 evaluate to $\bot$ in the original trail M.

In trail M_2, the added constraint eliminates the first interval of possible values for y and we can call the integer conflict explanation $\mathsf{explain}_{\mathbb{Z}}$ on M_2.[4] Again,

[4] By reducing the number of intervals we guarantee termination.

since the added assertion $(y \geq \lceil m \rceil)$ is necessary for explaining the inconsistency in M_2, the explanation clause for M_2 will be of the form

$$\mathsf{explain}_{\mathbb{Z}}(M_2, y) \equiv \neg(y \geq \lceil m \rceil) \vee D_2 \equiv (y \leq \lfloor m \rfloor) \vee D_2 \ \ .$$

Again, we know that all literals in the clause D_2 evaluate to $\bot$ in the original trail M.

In both cases above the explanation clauses are valid in the integers. We can therefore resolve the common literal with Boolean resolution and obtain a valid clause that we use as the final explanation

$$\mathsf{explain}_{\mathbb{Z}}(M, y) \equiv D_1 \vee D_2 \ \ .$$

Since literals of both D_1 and D_2 evaluate to $\bot$, the explanation clause is indeed a valid conflict clause for M being inconsistent with respect to y.

3.5 Main Algorithm

The core algorithm behind MCSat is based on the search-and-resolve loop common in modern SAT solvers (e.g. [11]). The main loop of the solver performs a direct search for a satisfying assignment and terminates either by finding an assignment that satisfies the original problem, or deduces that the problem is unsatisfiable. The main `check()` method is presented in Algorithm 1.

Algorithm 1: MCSat::check()

```
Data: solver trail $M$, relevant variables/terms to assign in $queue$
while true do
    propagate()
    if a plugin detected conflict and the conflict clause is $C$ then
        $R \leftarrow$ analyzeConflict($M$, $C$)
        if $R = \bot$ then return unsat
        backtrackWith($M$, $R$)
    else
        if $queue$.empty() then return sat
        $x \leftarrow queue$.pop()
        ownerOf($x$).decideValue($x$)
```

The search process goes forward, making continuous progress, either through propagation, conflict analysis, or by making a decision. The `propagate()` procedure invokes the (unit) propagation procedures provided by the enabled plugins. Each plugin is allowed to propagate new information to the top of the trail. If a plugin detects an inconsistency, it communicates the conflict to the solver by producing a conflict clause. This allows the solver to analyze the conflict using the `analyzeConflict()` procedure. If conflict analysis learns the empty clause

$\perp$, the problem is proved unsatisfiable. Otherwise the learned clause is used to backtrack the search, new relevant terms are collected from the learned clause R, and the search continues.

If the plugins have performed propagation to exhaustion, and no conflict was detected, the procedure makes progress by deciding a value for an unassigned variable. The solver picks an unassigned variable x to be assigned, and relegates the choice of the value to the plugin responsible for assigning x. A choice of value for the selected unassigned variable should exist, as otherwise some plugin should have detected the unit inconsistency. MCSat uses a uniform heuristic to select the next variable, regardless of its type. The heuristic is based on how often a variable is used in conflict resolution, and is popularly used in CDCL-style SAT solvers [26]. If all the relevant variables and terms are assigned to a value, then the trail is evaluation consistent and represents the satisfying assignment for the original problem.

4 Experiments

We have implemented the new method on top of NLSat in the MCSat framework within the YICES2 SMT solver [9].[5] For representation and advanced operations on polynomials, such as root isolation and explanation of conflicts through CAD projection, we rely on the LibPoly library.[6]

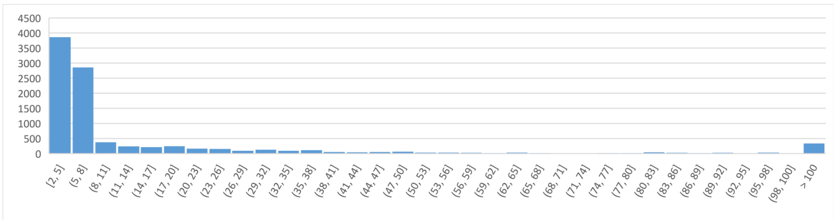

Fig. 5. Distribution of problems with respect to the number of integer variables involved (number of variables on the x axis, and number of problems with this number of variables on the y axis)

We have evaluated the new procedure on nonlinear integer benchmarks from the SMT-LIB library [3] (denoted as `QF_NIA` in the library). Most of the benchmarks in the library come from practical applications and are grouped into the following problem sets. The `AProVE` problems encode program termination conditions [13] and have between 2 and 985 integer variables. The `calypto` problem set contains problems relevant in hardware equivalence checking, with problems having between 3 and 50 integer variables. The `LassoRanker` problems encode

[5] Available at `http://yices.csl.sri.com/`.
[6] Available at `http://sri-csl.github.io/libpoly/`.

termination conditions of lasso-shaped programs and have between 15 and 35 integer variables. The set `LCTES` contains problems that involve integer division, with 673 integer variables each. The `leipzig` problems encode termination conditions for rewriting systems and have between 24 and 2606 integer variables. The `mcm` problems encode the MCM problem from [24], and have between 6 and 201 integer variables. The `UltimateAutomizer` set contains software verification queries from [16], with problems having between 4 and 144 integer variables. The `UltimateLassoRanker` problems encode non-termination of lasso-shaped programs [23], and have between 38 and 272 integer variables. To give an idea of the kinds of problems involved, Figure 5 presents distribution of number of problems by number of integer variables. The largest problems are in the `leipzig` set, with 10 problems with over 1000 integer variables and the largest problem having 2606 variables. [7]

To put the results in context, we compare yices2 with other state-of-the-art solvers that support nonlinear integer arithmetic, namely, aprove [13], smt-rat [6], and z3 [7]. The results are presented in Table 6. Each solver was run with a timeout of 40 minutes. Each column of the table corresponds to a different solver, and each row corresponds to a different problem set. For each problem set and solver combination we report the number of problems that the tool has solved, how many of the solved problems were unsatisfiable, and the total time (in seconds) that the tool took to solve those problems. For a more detailed comparison a scatter-plot of the performance of yices2 and other solvers is presented in Figure 7.[8]

Fig. 6. Experimental evaluation. Each row corresponds to a different problem set. Each column corresponds to a different solver. Each table entry shows the number of problems solved, the number of unsatisfiable problems solved, and the total time it took for the solved instances.

	yices2			aprove			smt-rat			z3		
problem set	solved	unsat	time (s)	solved	unsat	time (s)	solved	unsat	time (s)	solved	unsat	time (s)
AProVE (8829)	**8727**	**765**	**10197**	8028	0	7713	8251	223	9080	8310	285	22705
calypto (177)	**176**	**97**	**370**	77	0	1650	168	89	659	175	96	2517
LassoRanker (120)	101	97	664	3	0	5	24	21	9	**107**	**103**	**8065**
LCTES (2)	0	0	0	0	0	0	0	0	0	0	0	0
leipzig (167)	95	1	3541	162	0	3101	161	0	5075	**162**	**0**	**1080**
mcm (186)	12	0	5394	0	0	0	22	0	3622	**47**	**19**	**29368**
UltimateAutomizer (7)	**7**	**7**	**0**	0	0	0	1	1	2	**7**	**7**	**0**
UltimateLassoRanker (32)	**32**	**26**	**6**	6	0	16	30	24	118	32	26	20
	9150	**993**	**20172**	8276	0	12485	8657	358	18565	8840	536	63755

[7] All benchmarks are available at `http://smtlib.cs.uiowa.edu/`.

[8] For convenience, we've made the detailed results is available at `https://docs.google.com/spreadsheets/d/1Gu9PZMvgJ6dCjwXnTdggKRUP1uI6kz6lpI2dpWEsWVU`.

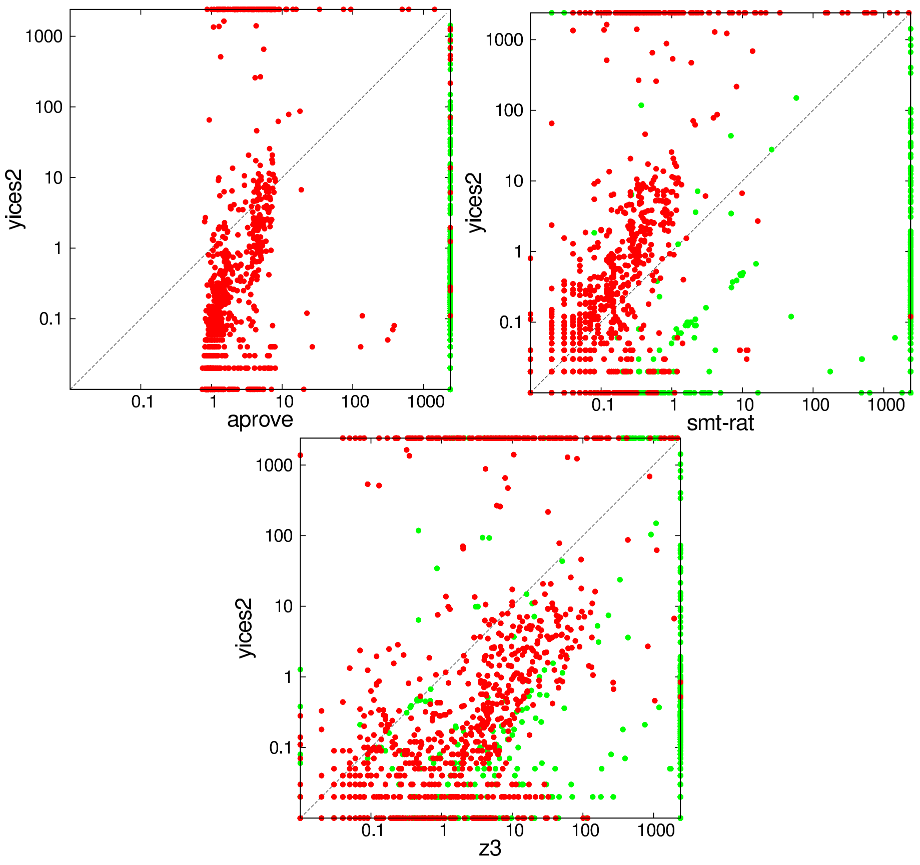

Fig. 7. Scatter-plot comparison of YICES2 and other solvers. Red points represent satisfiable problems, and green points represent unsatisfiable problems. Each axis (log scale) corresponds to the amount of time (in seconds) that each solver spent on the problem. Points below the $y = x$ line are the problems where YICES2 performs faster, and points on the top and right edges are problems where one of the solver ran out of time or terminated for other reasons.

As can be seen from these results, YICES2 is very efficient overall and solves the most problems from the SMT-LIB benchmarks. But, the new method truly excels on unsatisfiable problems, where it outperforms the other solvers by a significant margin. The real advantage of the new methods becomes apparent when solving problems (such as Example 1) where no combination of linear reasoning, interval reasoning, or bit-blasting can deduce unsatisfiablity (while other solvers fail on the problem, Example 1 is trivial for YICES2). The problems that YICES2 managed to show unsatisfiable contained as many as 468 integer variables, which is very encouraging considering the complexity of the underlying decision problem.

5 Conclusion

We have presented a new method for solving nonlinear integer problems based on the model-constructing approach to SMT and branch and bound. As opposed to existing methods that mostly rely on bit-blasting, the new method reasons directly in the integers and performs the branch-and-bound "splits" in a *conflict-driven* manner. The new method has been implemented in the YICES2 SMT solver and we have presented an extensive empirical evaluation where the new method is highly effective, and excels on unsatisfiable problems that can not be solved by other methods.

References

1. R. Albrecht, B. Buchberger, G. E. Collins, and R. Loos. *Computer algebra: symbolic and algebraic computation*, volume 4. 2012.
2. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177, 2011.
3. C. Barrett, A. Stump, and C. Tinelli. The satisfiability modulo theories library (SMT-LIB). *www.SMT-LIB.org*, 15:18–52, 2010.
4. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.
5. G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183, 1975.
6. F. Corzilius, G. Kremer, S. Junges, S. Schupp, and E. Ábrahám. SMT-RAT: An open source C++ toolbox for strategic and parallel SMT solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368, 2015.
7. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
8. L. De Moura and D. Jovanović. A model-constructing satisfiability calculus. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 1–12, 2013.

9. B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744, 2014.
10. B. Dutertre and L. De Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification*, pages 81–94, 2006.
11. N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and applications of satisfiability testing*, pages 502–518, 2004.
12. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354, 2007.
13. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *International Joint Conference on Automated Reasoning*, pages 184–191, 2014.
14. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.
15. A. Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation*, 8:1–27, 2012.
16. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52, 2013.
17. D. Jovanović. *SMT Beyond DPLL(T): A New Approach to Theory Solvers and Theory Combination*. PhD thesis, Courant Institute of Mathematical Sciences New York, 2012.
18. D. Jovanović, C. Barrett, and L. De Moura. The design and implementation of the model constructing satisfiability calculus. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 173–180, 2013.
19. D. Jovanović and L. De Moura. Solving non-linear arithmetic. In *International Joint Conference on Automated Reasoning*, pages 339–354, 2012.
20. D. Jovanović and L. De Moura. Cutting to the chase: Solving linear integer arithmetic. *Journal of automated reasoning*, 51(1):79–108, 2013.
21. T. King. *Effective Algorithms for the Satisfiability of Quantifier-Free Formulas Over Linear Real and Integer Arithmetic*. PhD thesis, Courant Institute of Mathematical Sciences New York, 2014.
22. G. Kremer, F. Corzilius, and E. Ábrahám. A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In *International Workshop on Computer Algebra in Scientific Computing*, pages 315–335, 2016.
23. J. Leike and M. Heizmann. Geometric series as nontermination arguments for linear lasso programs. In *14th International Workshop on Termination*, page 55.
24. N. P. Lopes, L. Aksoy, V. Manquinho, and J. Monteiro. Optimally solving the MCM problem using pseudo-boolean satisfiability. 2011.
25. Y. V. Matiyasevich. Hilbert's tenth problem. 1993.
26. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.
27. C. H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.
28. G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. *Machine intelligence*, 4:135–150, 1969.

29. A. Tarski. A decision method for elementary algebra and geometry. Technical Report R-109, Rand Corporation, 1951.
30. V. Weispfenning. Quantifier elimination for real algebra – the quadratic case and beyond. *Applicable Algebra in Engineering, Communication and Computing*, 8(2):85–101, 1997.