Experimenting on Solving Nonlinear Integer Arithmetic with Incremental Linearization*

Alessandro Cimatti[1], Alberto Griggio[1], Ahmed Irfan[1,2], Marco Roveri[1], and Roberto Sebastiani[2]

[1] Fondazione Bruno Kessler, Italy,
[lastname]@fbk.eu,
[2] DISI, University of Trento, Italy,
[firstname].[lastname]@unitn.it

Abstract. Incremental linearization is a conceptually simple, yet effective, technique that we have recently proposed for solving SMT problems over nonlinear real arithmetic constraints. In this paper, we show how the same approach can be applied successfully also to the harder case of nonlinear integer arithmetic problems. We describe in detail our implementation of the basic ideas inside the MathSAT SMT solver, and evaluate its effectiveness with an extensive experimental analysis over all nonlinear integer benchmarks in SMT-LIB. Our results show that MathSAT is very competitive with (and often outperforms) state-of-the-art SMT solvers based on alternative techniques.

1 Introduction

The field of Satisfiability Modulo Theories (SMT) has seen tremendous progress in the last decade. Nowadays, powerful and effective SMT solvers are available for a number of quantifier-free theories[3] and their combinations, such as equality and uninterpreted functions (UF), bit-vectors (BV), arrays (AX), and linear arithmetic over the reals (LRA) and the integers (LIA). A fundamental challenge is to go beyond the linear case, by introducing nonlinear polynomials – theories of nonlinear arithmetic over the reals (NRA) and the integers (NIA). Although the expressive power of nonlinear arithmetic is required by many application domains, dealing with nonlinearity is a very hard challenge. Going from SMT(LRA) to SMT(NRA) yields a complexity gap that results in a computational barrier in practice – most available complete solvers rely on Cylindrical Algebraic Decomposition (CAD) techniques [8], which require double exponential time in worst case. Adding integrality constraints exacerbates the problem even further, because reasoning on NIA has been shown to be undecidable [16].

Recently, we have proposed a conceptually simple, yet effective approach for dealing with the quantifier-free theory of nonlinear arithmetic over the reals,

* This work was funded in part by the H2020-FETOPEN-2016-2017-CSA project SC^2 (712689).

[3] In the following, we only consider quantifier-free theories, and we abuse the accepted notation by omitting the "QF_" prefix in the names of the theories.

called *Incremental Linearization* [4–6]. Its underlying idea is that of trading the use of expensive, exact solvers for nonlinear arithmetic for an abstraction-refinement loop on top of much less expensive solvers for linear arithmetic and uninterpreted functions. The approach is based on an abstraction-refinement loop that uses SMT(UFLRA) as abstract domain. The uninterpreted functions are used to model nonlinear multiplications, which are incrementally axiomatized, by means of linear constraints, with a lemma-on-demand approach.

In this paper, we show how incremental linearization can be applied successfully also to the harder case of nonlinear integer arithmetic problems. We describe in detail our implementation of the basic ideas, performed within the MathSAT [7] SMT solver, and evaluate its effectiveness with an extensive experimental analysis over all NIA benchmarks in SMT-LIB. Our results show that MathSAT is very competitive with (and often outperforms) state-of-the-art SMT solvers based on alternative techniques.

Related work. Several SMT solvers supporting nonlinear integer arithmetic (e.g, Z3 [10], SMT-RAT [9]) rely on the bit-blasting approach [12], in which a nonlinear integer satisfiability problem is iteratively reduced to a SAT problem by first bounding the integer variables, and then encoding the resulting problem into SAT. If the SAT problem is unsatisfiable then the bounds on the integer variables are increased, and the process is repeated. This approach is geared towards finding models, and it cannot prove unsatisfiability unless the problem is bounded.

In [3], the SMT(NIA) problem is approached by reducing it to SMT(LIA) via linearization. The linearization is performed by doing case analysis on the variables appearing in nonlinear monomials. Like the bit-blasting approach, the method aims at detecting satisfiable instances. If the domain of the problem is bounded, the method generates an equisatisfiable linear SMT formula. Otherwise, it solves a bounded problem and incrementally increases the bounds of some (heuristically chosen) variables until it finds a solution to the linear problem. In some cases, it may also detect (based on some heuristic) the unsatisfiability of the original problem.

The CVC4 [1] SMT solver uses a hybrid approach, in which a variant of incremental linearization (as presented in [5, 17]) is combined with bit-blasting.

Recent works presented in [13] and [15] have considered a method that combines solving techniques for SMT(NRA) with branch and bound. The main idea is to relax the NIA problem by interpreting the variables over the reals, and apply NRA techniques for solving it. Since the relaxed problem is an over-approximation of the original problem, the unsatisfiability of the NIA problem is implied by the unsatisfiability of the NRA problem. If the NRA-solver finds a non-integral solution a to a variable x, then a lemma $(x \leq \lfloor a \rfloor \vee x \geq \lceil a \rceil)$ is added to the NRA problem. Otherwise, an integral solution is found for the NIA problem. In [13], the Cylindrical Algebraic Decomposition (CAD) procedure (as presented in [14]) is combined with branch and bound in the MCSAT framework. This is the method used by the Yices [11] SMT solver. In [15], the authors

show how to combine CAD and virtual substitution with the branch-and-bound method in the DPLL(T) framework.

Contributions. Compared to our previous works on incremental linearization [4–6], we make the following contributions. First, we give a significantly more detailed description of our implementation (in the SMT solver MATHSAT), showing pseudo-code for all its major components. Second, we evaluate the approach over NIA problems, both by comparing it with the state of the art, and by evaluating the contributions of various components/heuristics of our procedure to its overall performance.

Structure of the paper. This paper is organized as follows. In §2 we provide some background on the ideas of incremental linearization. In §3 we describe our implementation in detail. In §4 we present our experimental evaluation. Finally, in §5 we draw some conclusions and outline directions for future work.

2 Background

We assume the standard first-order quantifier-free logical setting and standard notions of theory, satisfiability, and logical consequence.

We denote with $\mathbb{Z}$ the set of integer numbers. A *monomial* in variables $v_1, v_2, \ldots, v_n$ is a product $v_1^{\alpha_1} * v_2^{\alpha_2} * \ldots * v_n^{\alpha_n}$, where each α_i is a non-negative integer called exponent of the variable v_i. When clear from context, we may omit the multiplication symbol $*$ and simply write $v_1^{\alpha_1} v_2^{\alpha_2} \ldots v_n^{\alpha_n}$. A *polynomial* p is a finite linear combination of monomials with coefficients in $\mathbb{Z}$, i.e., $p \stackrel{\text{def}}{=} \Sigma_{i=0}^{n} c_i m_i$ where each $c_i \in \mathbb{Z}$ and each m_i is a monomial. A *polynomial constraint* or simply *constraint* P is of the form $p \bowtie 0$ where p is a polynomial and $\bowtie \in \{<, \leq, >, \geq\}$.[4]

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some theory or combination of theories. Most SMT solvers are based on the lazy/DPLL(T) approach [2], where a SAT solver is tightly integrated with a T-solver, that is demanded to decide the satisfiability of a list of constraints (treated as a conjunction of constraints) in the theory T. There exist several theories that the modern SMT solvers support. In this work we are interested in the following theories: *Equality and Uninterpreted Functions* (UF), *Linear Arithmetic* and *Nonlinear Arithmetic* over the integers (LIA and NIA, resp.), and in their combinations thereof.

We denote formulas with φ, lists of constraints with ϕ, terms with t, variables with v, constants with a, b, c, monomials with w, x, y, z, polynomials with p, functions with f, each possibly with subscripts. If μ is a model and v is a variable, we write $\mu[v]$ to denote the value of v in μ, and we extend this notation to terms

[4] In the rest of the paper, for simplifying the presentation we assume that an equality constraint is written as a conjunction of weak inequality constraints, and an inequality constraint is written as a disjunction of strict inequality constraints.

Basic:	*Sign:*	$v_1 * v_2 = (-v_1 * -v_2)$
		$v_1 * v_2 = -(-v_1 * v_2)$
		$v_1 * v_2 = -(v_1 * -v_2)$
	Zero:	$(v_1 = 0 \vee v_2 = 0) \leftrightarrow v_1 * v_2 = 0$
		$((v_1 > 0 \wedge v_2 > 0) \vee (v_1 < 0 \wedge v_2 < 0)) \leftrightarrow v_1 * v_2 > 0$
		$((v_1 < 0 \wedge v_2 > 0) \vee (v_1 > 0 \wedge v_2 < 0)) \leftrightarrow v_1 * v_2 < 0$
	Neutral:	$(v_1 = 1 \vee v_2 = 0) \leftrightarrow v_1 * v_2 = v_2$
		$(v_2 = 1 \vee v_1 = 0) \leftrightarrow v_1 * v_2 = v_1$
	Proportionality:	$\lvert v_1 * v_2\rvert \geq \lvert v_2\rvert \leftrightarrow (\lvert v_1\rvert \geq 1 \vee v_2 = 0)$
		$\lvert v_1 * v_2\rvert \leq \lvert v_2\rvert \leftrightarrow (\lvert v_1\rvert \leq 1 \vee v_2 = 0)$
		$\lvert v_1 * v_2\rvert \geq \lvert v_1\rvert \leftrightarrow (\lvert v_2\rvert \geq 1 \vee v_1 = 0)$
		$\lvert v_1 * v_2\rvert \leq \lvert v_1\rvert \leftrightarrow (\lvert v_2\rvert \leq 1 \vee v_1 = 0)$
Order:		$(v_1 * v_2 \bowtie v_3 \wedge v_4 > 0) \rightarrow v_1 * v_2 * v_4 \bowtie v_3 * v_4$
		$(v_1 * v_2 \bowtie v_3 \wedge v_4 < 0) \rightarrow v_3 * v_4 \bowtie v_1 * v_2 * v_4$
Monotonicity:		$(\lvert v_1\rvert \leq \lvert v_2\rvert \wedge \lvert v_3\rvert \leq \lvert v_4\rvert) \rightarrow \lvert v_1 * v_3\rvert \leq \lvert v_2 * v_4\rvert$
		$(\lvert v_1\rvert < \lvert v_2\rvert \wedge \lvert v_3 \leq \lvert v_4\rvert \wedge v_4 \neq 0) \rightarrow \lvert v_1 * v_3\rvert < \lvert v_2 * v_4\rvert$
		$(\lvert v_1\rvert \leq \lvert v_2\rvert \wedge \lvert v_3\rvert < \lvert v_4\rvert \wedge v_2 \neq 0) \rightarrow \lvert v_1 * v_3\rvert < \lvert v_2 * v_4\rvert$
Tangent plane:		$v_1 = a \rightarrow v_1 * v_2 = a * v_2$
		$v_2 = b \rightarrow v_1 * v_2 = b * v_1$
		$(v_1 > a \wedge v_2 < b) \rightarrow v_1 * v_2 < b * v_1 + a * v_2 - a * b$
		$(v_1 < a \wedge v_2 > b) \rightarrow v_1 * v_2 < b * v_1 + a * v_2 - a * b$
		$(v_1 < a \wedge v_2 < b) \rightarrow v_1 * v_2 > b * v_1 + a * v_2 - a * b$
		$(v_1 > a \wedge v_2 > b) \rightarrow v_1 * v_2 > b * v_1 + a * v_2 - a * b$

Fig. 1. Axioms of the multiplication function.

and formulas in the usual way. If ϕ is a list of constraints, we write $\bigwedge \phi$ to denote the formula obtained by taking the conjunction of all its elements.

We call a monomial m a *toplevel monomial* in a polynomial $p \stackrel{\text{def}}{=} \Sigma_{i=0}^{n} c_i m_i$ if $m = m_j$ for $0 \leq j \leq n$. Similarly, a monomial m is a *toplevel monomial* in φ if there exists a polynomial p in φ such that m is a toplevel monomial in p. Given φ, we denote with $\widehat{\varphi}$ the formula obtained by replacing every nonlinear multiplication between two monomials $x * y$ occurring in φ by a binary uninterpreted function $f_*(x, y)$.

We assume that the polynomials in φ are normalized by applying the distributivity property of multiplication over addition, and by sorting both the monomials and the variables in each monomial using a total order (e.g. lexicographic). Moreover, we always rewrite negated polynomial constraints into negation-free polynomial constraints by pushing the negation to the arithmetic relation (e.g., we write $\neg(p \leq 0)$ as $(p > 0)$).

```
result CHECK-NIA (φ : constraint list):
 1.  res = CHECK-UFLIA(φ̂):
 2.  if RES-IS-FALSE(res):
 3.      return res
 4.  μ = RES-GET-MODEL (res)
 5.  to_refine = ∅
 6.  φ' = {c | c ∈ φ and EVAL-MODEL(μ, c) = ⊥}
 7.  for each x * y in φ':
 8.      if EVAL-MODEL(μ, x * y) ≠ μ[x * y]^:
 9.          to_refine = to_refine ∪ {x * y}
10.  if to_refine = ∅:
11.      return ⟨TRUE, μ⟩
12.  res = CHECK-SAT(φ, μ)
13.  if RES-IS-TRUE(res):
14.      return res
15.  lemmas = ∅
16.  for round in ⟨1, 2, 3⟩:
17.      for each x * y in to_refine:
18.          L = GENERATE-LEMMAS(x * y, μ, round, to_refine, φ)
19.          lemmas = lemmas ∪ L
20.      if lemmas ≠ ∅:
21.          return ⟨UNDEF, lemmas⟩
22.  return ⟨UNKNOWN⟩
```

Fig. 2. The top-level NIA theory solver procedure.

Overview of incremental linearization. The main idea of incremental linearization is to trade the use of expensive, exact solvers for nonlinear arithmetic for an abstraction-refinement loop on top of much less expensive solvers for linear arithmetic and uninterpreted functions. First, the input SMT(NIA) formula φ is abstracted to the SMT(UFLIA) formula $\widehat{\varphi}$ (called its UFLIA-abstraction). Then the loop begins by checking the satisfiability of $\widehat{\varphi}$. If the SMT(UFLIA) check returns false then the input formula is unsatisfiable. Otherwise, the model μ for $\widehat{\varphi}$ is used to build an UFLIA underapproximation $\widehat{\varphi}^*$ of φ, with the aim of finding a model for the original NIA formula φ. If the SMT check for $\widehat{\varphi}^*$ is satisfiable, then φ is also satisfiable. Otherwise, a conjunction of linear *lemmas* that is sufficient to rule out the spurious model μ is added to $\widehat{\varphi}$, thus improving the precision of the abstraction, and another iteration of the loop is performed. The lemmas added are instances of the axioms of Fig. 1 obtained by replacing the free variables with terms occurring in φ, selected among those that evaluate to false under the current spurious model μ.

3 Implementing incremental linearization in a lazy SMT solver

We now describe in detail our implementation of the basic incremental linearization ideas as a theory solver inside an SMT prover based on the lazy/DPLL(T) approach. The pseudo-code for the toplevel algorithm is shown in Fig. 2. The

```
value EVAL-MODEL (μ : model, t : term):
1.  match t with
2.      x ⋈ y:      return (EVAL-MODEL(μ, x) ⋈ EVAL-MODEL(μ, y) ? ⊤ : ⊥)
3.      x ∗ y:      return EVAL-MODEL(μ, x) ∗ EVAL-MODEL(μ, y)
4.      x + y:      return EVAL-MODEL(μ, x) + EVAL-MODEL(μ, y)
5.      c ∗ x:      return c ∗ EVAL-MODEL(μ, x)
6.      v :         return μ[v]
7.      c :         return c
```

Fig. 3. Recursive model evaluation.

```
result CHECK-SAT (φ : constraint list, μ : UFLIA-model):
1.  φ = ⋀ φ
2.  for each x ∗ y in φ:
3.      c_x = EVAL-MODEL(μ, x)
4.      c_y = EVAL-MODEL(μ, y)
5.      φ = φ ∧ ((x ∗ y = c_x ∗ y ∧ x = c_x) ∨ (x ∗ y = c_y ∗ x ∧ y = c_y))
6.  return SMT-UFLIA-SOLVE (φ̂)
```

Fig. 4. Searching for a model via linearization.

algorithm takes as input a list of constraints ϕ, corresponding to the NIA constraints in the partial assignment that is being explored by the SAT search, and it returns a result consisting of a status flag plus some additional information that needs to be sent back to the SAT solver. If the status is TRUE, then ϕ is satisfiable, and a model μ for it is also returned. If the status is FALSE, then ϕ is unsatisfiable, and a conflict set $\phi' \subseteq \phi$ (serving as an explanation for the inconsistency of ϕ) is also returned. If the status is UNDEF, the satisfiability of ϕ cannot be determined yet. In this case, the returned result contains also a set of lemmas to be used by the SAT solver to refine its search (i.e. those lemmas are learnt by the SAT solver, and the search is resumed). Finally, a status of UNKNOWN means that the theory solver can neither determine the satisfiability of ϕ nor generate additional lemmas [5]; in this case, the search is aborted.

CHECK-NIA starts by invoking a theory solver for UFLIA on the abstract version $\widehat{\phi}$ of the input problem (lines 1–4), in which all nonlinear multiplications are treated as uninterpreted functions. The unsatisfiability of $\widehat{\phi}$ immediately implies that ϕ is inconsistent. Otherwise, the UFLIA solver generates a model μ for $\widehat{\phi}$. μ is then used (lines 5–9) to determine the set of nonlinear multiplications that need to be refined. This is done by collecting all nonlinear multiplication terms $x * y$ which have a wrong value in μ; that is, for which the value of the abstraction $\widehat{x * y}$ is different from the value obtained by fully evaluating the multiplication under μ (using the EVAL-MODEL function shown in Fig. 3). It is important to observe that here we can limit the search for multiplications to refine only to those that appear in constraints that evaluate to false under μ

[5] This can happen when the tangent lemmas (see Fig. 8) are not used.

```
lemma set GENERATE-LEMMAS (x ∗ y : term, μ : model, r : int, to_refine : term set,
                           φ : constraint list):
1.  if r = 1:
2.      return GENERATE-BASIC-LEMMAS (x ∗ y, μ)
3.  else:
4.      if r = 2:
5.          L = GENERATE-ORDER-LEMMAS(x ∗ y, μ, φ)
6.          if L ≠ ∅:
7.              return L
8.          toplevel = true
9.      else:
10.         toplevel = false
11.     L = GENERATE-MONOTONICITY-LEMMAS(x ∗ y, μ, to_refine, toplevel)
12.     if L ≠ ∅:
13.         return L
14.     return GENERATE-TANGENT-LEMMAS(x ∗ y, μ, toplevel)
```

Fig. 5. Main lemma generation procedure.

(line 6). In fact, if all the constraints evaluate to true, then by definition μ is a model for them, and we can immediately conclude that ϕ is satisfiable (line 10).

Even when μ is spurious, it can still be the case that there exists a model for ϕ that is "close" to μ. This is the idea behind the CHECK-SAT procedure of Fig. 4, which uses μ as a guide in the search for a model of ϕ. CHECK-SAT works by building an UFLIA-underapproximation of ϕ, in which all multiplications are forced to be linear. The resulting formula $\widehat{\varphi}$ can then be solved with an SMT solver for UFLIA. Although clearly incomplete, this procedure is cheap (since the Boolean structure of $\widehat{\varphi}$ is very simple) and, as our experiments will show, surprisingly effective.

When CHECK-SAT fails, we proceed to the generation of lemmas for refining the spurious model μ (lines 15–21). Our lemma generation strategy works in three rounds: we invoke the GENERATE-LEMMAS function (Fig. 5) on the multiplication terms $x * y$ that need to be refined using increasing levels of effort, stopping at the earliest successful round – i.e., a round in which lemmas are generated. In the first round, only basic lemmas encoding simple properties of multiplications (sign, zero, neutral, proportionality in Fig. 1) are considered (GENERATE-BASIC-LEMMAS). In the second round, we consider also "order" lemmas (GENERATE-ORDER-LEMMAS, Fig. 6), i.e. lemmas obtained via (a restricted, model-driven) application of the order axioms of $\mathbb{Z}$. If GENERATE-ORDER-LEMMAS fails, we proceed to generating monotonicity (GENERATE-MONOTONICITY-LEMMAS, Fig. 7) and tangent plane (GENERATE-TANGENT-LEMMAS, Fig. 8) lemmas, restricting the instantiation however to only toplevel monomials. Finally, in the last round, we repeat the generation of monotonicity and tangent lemmas, considering this time also non-toplevel monomials.

```
lemma set GENERATE-ORDER-LEMMAS (x ∗ y : term, μ : model, φ : constraint list):
1.  for each variable v in x ∗ y:
2.      monomials = GET-MONOMIALS (v, φ)
3.      bounds = GET-BOUNDS (v, φ)
4.      for each (w ∗ v ⋈ p) in bounds:
5.          for each t ∗ v in monomials:
6.              if t ∗ w ∗ v in φ and  t ∗ p in φ:
7.                  n = EVAL-MODEL(t)
8.                  if n = 0:
9.                      continue
10.                 else if n > 0:
11.                     ψ = ((w ∗ v ⋈ p) ∧ t > 0) → (t ∗ w ∗ v ⋈ t ∗ p)
12.                 else:
13.                     ψ = ((w ∗ v ⋈ p) ∧ t < 0) → (t ∗ p ⋈ t ∗ w ∗ v)
14.                 if EVAL-MODEL(μ, ψ) = ⊥:
15.                     return {ψ}
16. return ∅
```

Fig. 6. Generation of order lemmas.

Lemma generation procedures. We now describe our lemma generation procedures in detail. The pseudo-code is reported in Figures 5–8. All procedures share the following two properties: (i) the lemmas generated do not contain any nonlinear multiplication term that was not in the input constraints ϕ; and (ii) all the generated lemmas evaluate to false ($\bot$) in the current model μ.

The function GENERATE-BASIC-LEMMAS, whose pseudo-code is not reported, simply instantiates all the basic axioms for the input term $x * y$ that satisfy points (i) and (ii) above.

The function GENERATE-ORDER-LEMMAS (Fig. 6) uses the current model and asserted constraints to produce lemmas that are instances of the order axiom for multiplication. It is based on ideas that were first implemented in the CVC4 [1] SMT solver.[6] It works by combining, for each variable v in the input term $x * y$, the monomials $t * v$ in which v occurs (retrieved by GET-MONOMIALS) with the predicates of the form $(w * v \bowtie p)$ (where $\bowtie \in \{<, >, \leq, \geq\}$ and p is a polynomial) that are induced by constraints in ϕ (which are collected by the GET-BOUNDS function). The (non-constant) coefficient t of v in the monomial $t * v$ is used to generate the terms $t * w * v$ and $t * p$: if both occur[7] in the input constraints ϕ, then an instance of the order axiom is produced, using the current model μ as a guide (lines 7–15).

The function GENERATE-MONOTONICITY-LEMMAS (Fig. 7) returns instances of monotonicity axioms relating the current input term $x * y$ with other monomials that occur in the set of terms to refine. In the second round of lemma generation, only toplevel monomials are considered.

[6] We are grateful to Andrew Reynolds for fruitful discussions about this.

[7] It is important to stress here that we keep the monomials in a normal form by reordering their variables, although this is not shown explicitly in the pseudo-code.

```
lemma set GENERATE-MONOTONICITY-LEMMAS (x ∗ y : term, μ : model, to_refine : term set,
                                        toplevel : bool):
1.  if toplevel ≠ IS-TOPLEVEL-MONOMIAL(x ∗ y):
2.      return ∅
3.  L = ∅
4.  for each w ∗ z in to_refine:
5.      if not toplevel or IS-TOPLEVEL-MONOMIAL (z ∗ w):
6.          ψ1 = (|x| ≤ |w| ∧ |y| ≤ |z|) → |x ∗ y| ≤ |w ∗ z|
7.          ψ2 = (|x| ≤ |z| ∧ |y| ≤ |w|) → |x ∗ y| ≤ |w ∗ z|
8.          ψ3 = (|x| < |w| ∧ |y| ≤ |z| ∧ z ≠ 0) → |x ∗ y| < |w ∗ z|
9.          ψ4 = (|x| < |z| ∧ |y| ≤ |w| ∧ w ≠ 0) → |x ∗ y| < |w ∗ z|
10.         ψ5 = (|x| ≤ |w| ∧ |y| < |z| ∧ w ≠ 0) → |x ∗ y| < |w ∗ z|
11.         ψ6 = (|x| ≤ |z| ∧ |y| < |w| ∧ z ≠ 0) → |x ∗ y| < |w ∗ z|
12.         L = L ∪ {ψi | EVAL-MODEL(μ, ψi) = ⊥}
13. return L
```

Fig. 7. Generation of monotonicity lemmas.

Finally, the function GENERATE-TANGENT-LEMMAS (Fig. 8) produces instances of the tangent plane axioms. In essence, the function instantiates all the clauses of the tangent plane lemma using the two factors x and y of the input multiplication term $x * y$ and their respective values a and b in μ, returning all the instances that are falsified by μ. This is done in lines 15–21 of Fig. 8. In our actual implementation, however, we do not use the model values a and b directly to generate tangent lemmas, but we instead use a heuristic that tries to reduce the number of tangent lemmas generated for each $x * y$ term to refine. More specifically, we keep a 4-value tuple $\langle l_x, l_y, u_x, u_y \rangle$ associated with each $x*y$ term in the input problem (which we call *frontier*) consisting of the smallest and largest of the previous model values for x and y for which a tangent lemma has been generated, and for each frontier we maintain an invariant that whenever x is in the interval $[l_x, u_x]$ or y is in the interval $[l_y, u_y]$, then $x * y$ has both an upper and a lower bound. This condition is achieved by adding tangent lemmas for the following four points of each frontier: $(l_x, l_y), (l_x, u_y), (u_x, l_y), (u_x, u_y)$ (the function UPDATE-TANGENT-FRONTIER in Fig. 8 generates those lemmas). If the current model values a and b for x and y are outside the intervals $[l_x, u_x]$ and $[l_y, u_y]$ respectively, we try to adjust them with the goal of enlarging the frontier as much as possible whenever we generate a tangent plane. Intuitively, this can be seen as a form of lemma generalisation. The procedure is shown in lines 6–14 of Fig. 8: the various PUSH-TANGENT-POINT* functions try to move the input points along the specified directions (either 'U'p, by increasing a value, or 'D'own, by decreasing it) as long as the tangent plane passing through (a, b) still separates the multiplication curve from the spurious value c.[8]

[8] In our implementation we use a bounded (dichotomic) search for this. For example, for the 'UU' direction we try increasing both a and b until either the tangent plane passing through (a, b) cannot separate the multiplication curve from the bad point c anymore, or we reach a maximum bound on the number of iterations.

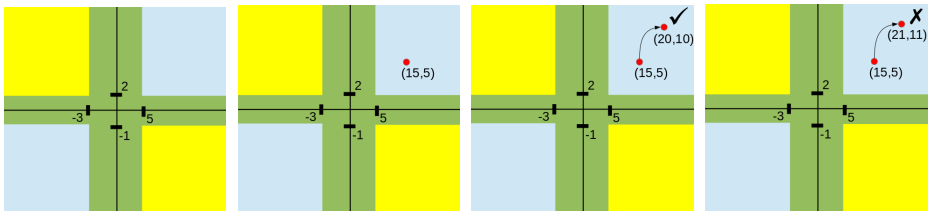
```
lemma set GENERATE-TANGENT-LEMMAS (x * y : term, μ : model, toplevel : bool):
 1.  if toplevel ≠ IS-TOPLEVEL-MONOMIAL(x * y):
 2.      return ∅
 3.  a = EVAL-MODEL(μ, x)
 4.  b = EVAL-MODEL(μ, y)
 5.  c = μ[x * y]
 6.  l_x, l_y, u_x, u_y = GET-TANGENT-FRONTIER(x * y)
 7.  if a < l_x and b < l_y: a, b = PUSH-TANGENT-POINTS-DD(x * y, a, b, c)
 8.  else if a < l_x and b > u_y: a, b = PUSH-TANGENT-POINTS-DU(x * y, a, b, c)
 9.  else if a > u_x and b > u_y: a, b = PUSH-TANGENT-POINTS-UU(x * y, a, b, c)
10.  else if a > u_x and b < l_y: a, b = PUSH-TANGENT-POINTS-UD(x * y, a, b, c)
11.  else if a < l_x: a = PUSH-TANGENT-POINT1-D(x * y, a, b, c)
12.  else if a > u_x: a = PUSH-TANGENT-POINT1-U(x * y, a, b, c)
13.  else if b < l_y: b = PUSH-TANGENT-POINT2-D(x * y, a, b, c)
14.  else if b > u_y: b = PUSH-TANGENT-POINT2-U(x * y, a, b, c)
15.  ψ1 = (x = a → x * y = a * y)
16.  ψ2 = (y = b → x * y = b * x)
17.  ψ3 = (x > a ∧ y < b) → (x * y < b * x + a * y − a * b)
18.  ψ4 = (x < a ∧ y > b) → (x * y < b * x + a * y − a * b)
19.  ψ5 = (x < a ∧ y < b) → (x * y > b * x + a * y − a * b)
20.  ψ6 = (x > a ∧ y > b) → (x * y > b * x + a * y − a * b)
21.  L = {ψ_i | EVAL-MODEL(μ, ψ_i) = ⊥}
22.  if L ≠ ∅:
23.      L = L ∪ UPDATE-TANGENT-FRONTIER(x * y, a, b)
24.  return L
```

Fig. 8. Generation of tangent lemmas.

Example 1 (Tangent frontier enlargement – Fig. 9). Let $\langle -3, -1, 5, 2\rangle$ be the current frontier of $x * y$ during the search. Suppose the abstract model gives: $\mu[x] = a = 15$, $\mu[y] = b = 5$, and $\mu[\widehat{x * y}] = c = 48$. This model is spurious because $15 * 5 \neq 48$. Notice that the point $(15, 5)$ is outside of the frontier, because 15 is not in $[-3, 5]$ and 5 is not in $[-1, 2]$. So, during the tangent lemmas generation, the function PUSH-TANGENT-POINTS-UU can return $a = 20$ and $b = 10$, as one of the constraints of the tangent lemma instantiated at that point is violated by the current model, i.e., we can obtain the following clauses from the tangent lemma:

$$x > 20 \wedge y < 10 \rightarrow x * y < 10 * x + 20 * y - 200$$
$$x < 20 \wedge y > 10 \rightarrow x * y < 10 * x + 20 * y - 200$$
$$x < 20 \wedge y < 10 \rightarrow x * y > 10 * x + 20 * y - 200$$
$$x > 20 \wedge y > 10 \rightarrow x * y > 10 * x + 20 * y - 200$$

by plugging in the values $x = 15$, $y = 5$, and $x * y = 48$, then we obtain a conflict in the third clause because $15 < 20$ and $5 < 10$, but $48 \not> 10 * 15 + 20 * 5 - 200$. This means that the tangent lemma instantiated at point $(20, 10)$ can be used for refinement (Fig. 9(c)).



(a) current frontier (b) original point (15,5) for the tangent lemma (c) successful pushed point (20,10) for the tangent lemma (d) unsuccessful pushed point (21,11) for the tangent lemma

Fig. 9. Illustration of the strategy for adjusting the refinement point for the tangent lemma.

However, if we use $(21, 11)$ for the tangent lemma instantiation, we get the following clauses:

$$x > 21 \wedge y < 11 \rightarrow x * y < 11 * x + 21 * y - 231$$
$$x < 21 \wedge y > 11 \rightarrow x * y < 11 * x + 21 * y - 231$$
$$x < 21 \wedge y < 11 \rightarrow x * y > 11 * x + 21 * y - 231$$
$$x > 21 \wedge y > 11 \rightarrow x * y > 11 * x + 21 * y - 231$$

Notice that, all these clauses are satisfied if we plug in the values $x = 15$, $y = 5$, and $x * y = 48$. Therefore, we cannot use them for refinement (Fig. 9(d)).

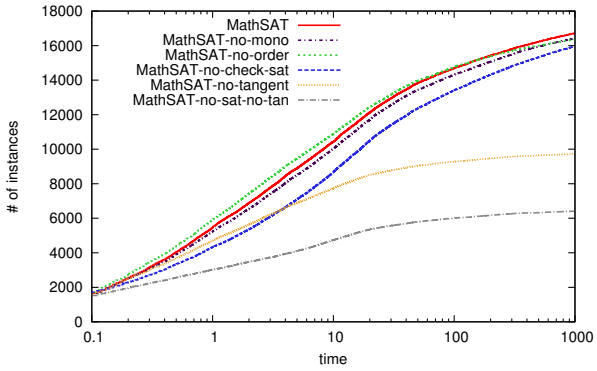
4 Experimental Analysis

We have implemented our incremental linearization procedure in our SMT solver MathSAT [7]. In this section, we experimentally evaluate its performance. Our implementation and experimental data are available at `https://es.fbk.eu/people/irfan/papers/sat18-data.tar.gz`.

Setup and Benchmarks. We have run our experiments on a cluster equipped with 2.6GHz Intel Xeon X5650 machines, using a time limit of 1000 seconds and a memory limit of 6 Gb.

For our evaluation, we have used all the benchmarks in the QF_NIA category of SMT-LIB [18], which at the present time consists of 23876 instances. All the problems are available from the SMT-LIB website.

Our evaluation is composed of two parts. In the first, we evaluate the contribution of different parts of our procedure to the overall performance of MathSAT, by comparing different configurations of the solver. In the second part, we compare our best configuration against the state of the art in SMT solving for nonlinear integer arithmetic.



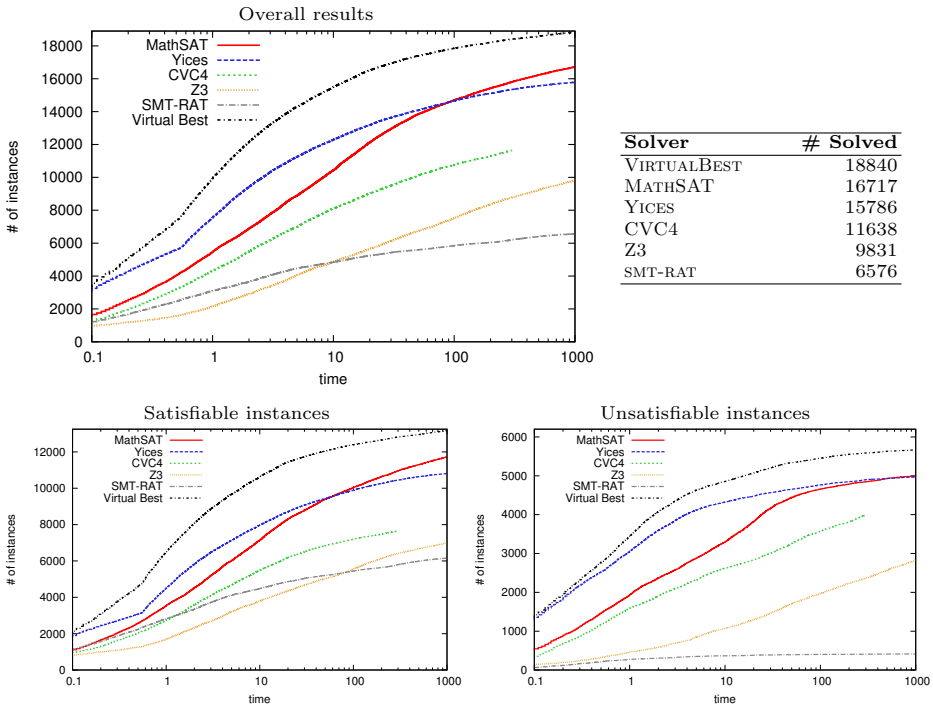
Solver	# Solved
MathSAT	16717
MathSAT-no-mono	16416
MathSAT-no-order	16379
MathSAT-no-check-sat	15971
MathSAT-no-tangent	9731
MathSAT-no-check-sat-no-tangent	6409

Fig. 10. Comparison among different configurations of MathSAT.

Comparison of different configurations. We evaluate the impact of the main components of our procedure, by comparing five different configurations of MathSAT:

- The standard configuration, using all the components described in the previous section (simply denoted MathSAT);
- a configuration with check-sat disabled (denoted MathSAT-no-check-sat);
- a configuration with generate-order-lemmas disabled (denoted MathSAT-no-order);
- a configuration with generate-monotonicity-lemmas disabled (denoted MathSAT-no-mono);
- a configuration with generate-tangent-lemmas disabled (denoted MathSAT-no-tangent); and finally
- a configuration with both check-sat and generate-tangent-lemmas disabled (denoted MathSAT-no-check-sat-no-tangent).

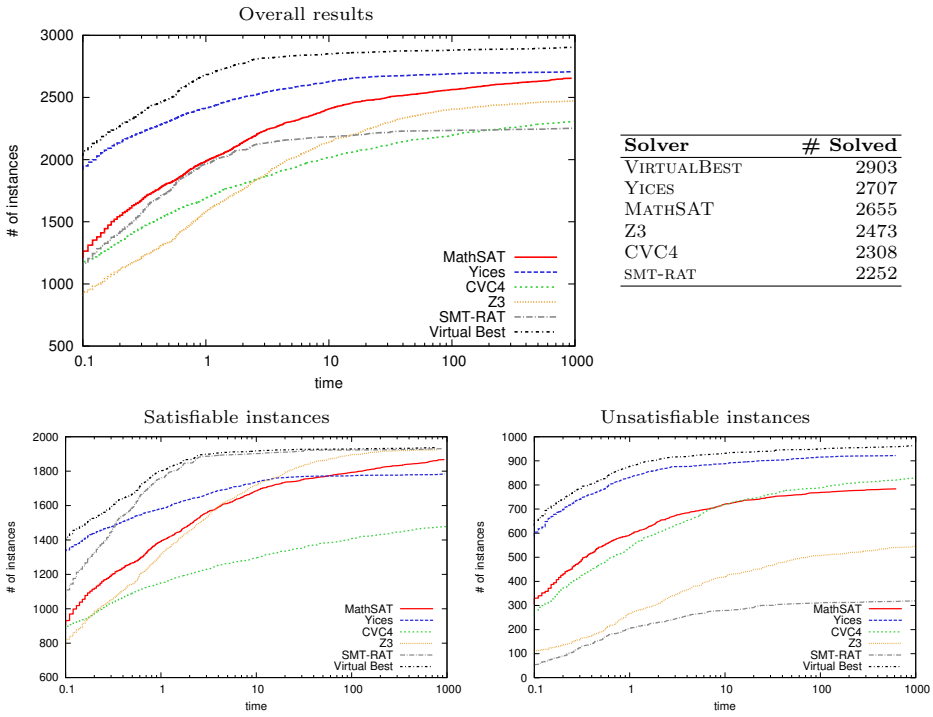
The results are presented in Fig. 10. The plot on the left shows, for each configuration, the number of instances that could be solved (on the y axis) within the given time limit (on the x axis). The table on the right shows the ranking of the configurations according to the number of instances solved. From Fig. 10, we can see that all components of our procedure contribute to the performance of MathSAT. As expected, tangent lemmas are crucial, but it is also interesting to observe that the cheap satisfiability check by linearization is very effective, leading to an overall performance boost and to the successful solution of 746 additional benchmarks that could not be solved by MathSAT-no-check-sat. Finally, although the addition of order axioms (by generate-order-lemmas) does not pay off for simpler problems, its impact is clearly visible for harder instances, allowing MathSAT to solve 338 more benchmarks than MathSAT-no-order.



Solver	# Solved
VirtualBest	18840
MathSAT	16717
Yices	15786
CVC4	11638
Z3	9831
SMT-RAT	6576

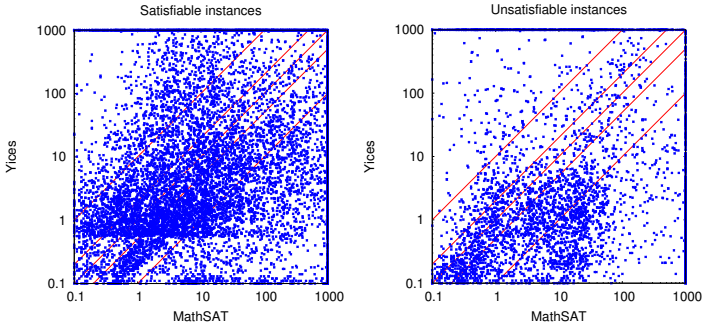
Fig. 11. Comparison with state-of-the-art SMT solvers for NIA.

Comparison with the state of the art. In the second part of our experiments, we compare MathSAT with the state-of-the-art SMT solvers for NIA. We consider CVC4 [1], SMT-RAT [9], Yices [11] and Z3 [10]. Fig. 11 and Fig. 12 show a summary of the results (with separate plots for satisfiable and unsatisfiable instances in addition to the overall plot), whereas Fig. 13 shows a more detailed comparison between MathSAT and Yices. Additional information about the solved instances for each benchmark family is given in Table 1. From the results, we can see that the performance of MathSAT is very competitive: not only it solves more instances than all the other tools, but it is also faster than CVC4, SMT-RAT and Z3. On the other hand, Yices is much faster than MathSAT in the majority of cases, especially on easy unsatisfiable instances (solved in less than 10 seconds). However, the two tools are very complementary, as shown by Fig. 13: MathSAT can solve 2436 instances for which Yices times out, whereas Yices can successfully handle 1505 instances that MathSAT is unable to solve. Moreover, MathSAT overall solves 931 more problems (915 satisfiable and 16 unsatisfiable) than Yices in the given resource limits.



Solver	# Solved
VirtualBest	2903
Yices	2707
MathSAT	2655
Z3	2473
CVC4	2308
SMT-RAT	2252

Fig. 12. Comparison with state-of-the-art SMT solvers for NIA – without the VeryMax benchmarks.



Solver	# Unique
MathSAT	2436
Yices	1505

Fig. 13. Detailed comparison between MathSAT and Yices.

5 Conclusions

We have presented a solver for satisfiability modulo nonlinear integer arithmetic based on the incremental linearization approach. Our empirical analysis of its performance over all the nonlinear integer benchmarks in the SMT-LIB library shows that the approach is very competitive with the state of the art: our solver MathSAT can solve many problems that are out of reach for other tools, and

Table 1. Summary of the comparison with the state of the art.

	Total (23876)	**AProVE** (2409)	**Calypto** (177)	**LassoRanker** (106)	**LCTES** (2)	**Leipzig** (167)	**MCM** (186)	**Ultimate Automizer** (7)	**Ultimate LassoRanker** (32)	**VeryMax** (20790)
MathSAT	**11723/4993**	1642/561	**79**/89	**4/100**	0/**1**	126/**2**	12/0	0/**7**	**6/26**	**9854/4207**
Yices	10808/4977	1595/**708**	**79/97**	**4**/84	0/0	92/1	8/0	0/**7**	**6/26**	9024/4054
CVC4	7653/3984	1306/608	77/89	**4**/94	0/**1**	84/**2**	6/0	0/6	**6/26**	6170/3158
Z3	6993/2837	1656/325	78/96	**4**/92	0/0	**162**/0	20/**1**	0/**7**	**6/26**	5067/2290
smt-rat	6161/414	**1663**/184	**79**/89	3/20	0/0	160/0	**21**/0	0/1	**6/26**	4229/94
VirtualBest	13169/5669	1663/724	79/97	4/101	0/1	162/2	23/1	0/7	6/26	11232/4710

Each column shows a family of benchmarks in the QF_NIA division of SMT-LIB. For each solver, the table shows the number of sat/unsat results in each family. The best performing tools (in terms of # of results) are reported in boldface.

overall it solves the highest number of instances. Our evaluation has however also shown that current approaches for SMT(NIA) are very complementary, with no tool that always outperforms all the others. This suggests the investigation of hybrid approaches that combine multiple methods as a very promising direction for future work.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. pp. 171–177 (2011)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
3. Borralleras, C., Lucas, S., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Sat modulo linear arithmetic for solving polynomial constraints. J. Autom. Reason. 48(1), 107–131 (Jan 2012)
4. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions. Under Submission (2017), available at `https://es.fbk.eu/people/irfan/papers/inclin-smt-vmt-nl-tf.pdf`
5. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In: TACAS 2017, Proceedings, Part I. pp. 58–75 (2017)
6. Cimatti, A., Griggio, A., Irfan, A., Roveri, M., Sebastiani, R.: Satisfiability modulo transcendental functions via incremental linearization. In: CADE. Lecture Notes in Computer Science, vol. 10395, pp. 95–113. Springer (2017)
7. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. LNCS, vol. 7795, pp. 93–107. Springer (2013)
8. Collins, G.E.: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition–preliminary Report. SIGSAM Bull. 8(3), 80–90 (Aug 1974)

9. Corzilius, F., Loup, U., Junges, S., Ábrahám, E.: SMT-RAT: An SMT-compliant nonlinear real arithmetic toolbox. In: SAT. pp. 442–448. Springer (2012)
10. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. pp. 337–340. Springer (2008)
11. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). LNCS, vol. 8559, pp. 737–744. Springer (July 2014)
12. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: SAT 2007, Proceedings. LNCS, vol. 4501, pp. 340–354. Springer (2007)
13. Jovanovic, D.: Solving nonlinear integer arithmetic with MCSAT. In: VMCAI 2017, Proceedings. LNCS, vol. 10145, pp. 330–346. Springer (2017)
14. Jovanović, D., De Moura, L.: Solving non-linear arithmetic. In: IJCAR. pp. 339–354. Springer (2012)
15. Kremer, G., Corzilius, F., Ábrahám, E.: A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In: CASC 2016, Proceedings. LNCS, vol. 9890, pp. 315–335. Springer (2016)
16. Matiyasevich, Y.V.: Hilbert's Tenth Problem. Foundations of computing, MIT Press (1993)
17. Reynolds, A., Tinelli, C., Jovanovic, D., Barrett, C.: Designing theory solvers with extensions. In: FroCoS. LNCS, vol. 10483. Springer (2017)
18. SMT-LIB, The Satisfiability Modulo Theories Library. `http://smtlib.org`