A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo

Sylvain Conchon, Mohamed Iguernelala, Alain Mebsout

A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo

Sylvain Conchon* Mohamed Iguernelala*,** Alain Mebsout*
*LRI, Université Paris-Sud, Orsay F-91405
**OCamlPro SAS, Gif-sur-Yvette F-91190

***Abstract*—In this paper, we describe a *collaborative* framework for reasoning modulo simple properties of non-linear integer arithmetic. This framework relies on the AC(X) combination method and on interval calculus. The first component is used to handle equalities of linear integer arithmetic and associativity and commutativity properties of non-linear multiplication. The interval calculus component is used — in addition to standard linear operations over inequalities — to refine bounds of non-linear terms and to inform the SAT solver about judicious case-splits on bounded intervals. The framework has been implemented in the Alt-Ergo theorem prover. We show its effectiveness on a set of formulas generated from deductive program verification.**

I. Introduction

Verification conditions that are produced by tools such as Boogie [3], Frama-C [7], Why3 [9], or SPARK Hi-Lite [26] sometimes contain formulas involving non-linear integer arithmetic. Non-linear operators (*e.g.* multiplication, Euclidean division, modulo) are usually generated from loop invariants or program specifications. For example, the verification of the following annotated C program computing the integer square root with the Newton method generates a proof obligation:

$$\forall n, p, r \in Z.\ n > 0 \ \wedge\ p = n \ \wedge\ r = \frac{n+1}{2} \implies r = \frac{p + \frac{n}{p}}{2}$$

```
/*@ requires n > 0;
    ensures  \result * \result <= n;
 */
int isqrt (int n) {
  int p = n;
  int r = (n + 1) / 2;
  /*@ loop invariant
      r > 0 && r = (p + n / p) / 2 */
  while (r != p) {
    p = r;
    r = (r + n / r) / 2;
  }
  return p;
}
```

While simple from a mathematical point of view, this is enough to block SMT solvers that are used as back-ends in these program verification frameworks. For instance, state-of-the art SMT solvers such as Z3 [17], CVC3 [6], CVC4 [4] and Yices [18] fail to prove the entirety of proof obligations from the previous example.

The particularity of these formulas is that they refer to *unbounded mathematical* integers and contain a combination of symbols from several theories (uninterpreted function symbols, arrays, records, enumerations, etc). Additionally, they are drowned among a context with hundreds of axioms describing values and operations of the programming language in first-order logic (data type representations, memory model, pointer arithmetic, etc).

Non-linear arithmetic over the *reals* (NRA) was shown to be decidable by Tarski [37]. Among the numerous decision procedures for NRA, *cylindrical algebraic decomposition* (CAD) [12] and *Gröbner bases* [27] are examples of complete methods. There also exist incomplete methods based on *interval constraint propagation* (ICP) [20], [38] and *virtual substitutions* (VS) [39] that are much more efficient in practice. These methods are implemented in different dedicated solvers like QEPCAD [11] and RAHD [34] or in general purpose SMT solvers like Z3 [17] which uses a combination of ICP and VS. NRA is still an active field of research in the SMT community [25].

Gödel showed that non-linear arithmetic over *integers* (NIA) is an undecidable problem [22]. While there exists a large panel of work on NRA, only very few incomplete methods exist for NIA. The most wildly used methods rely on a form of linearization or booleanization for bounded integers. Some approaches are based on bit-blasting [1], [35] which destroys high-level arithmetic constructs and are only applicable for integer variables living in limited ranges. Other techniques use encodings to SMT(BV) (bit-vectors) [2] or to SMT(LIA) [10], [21] with lazy bounding and refinements. These methods only handle *bounded* integers and work by exhibiting models in satisfiable instances but are not very well suited to prove unsatisfiability. An other related field of research concerns *diophantine* (in)equations, a restricted form of non-linear integer constraints [36]. While this problem has also been shown to be undecidable [32], there exist some complete algorithms when the degree of polynomials is bounded [15].

Following the remark that only a handful of simple non-linear arithmetic constraints arise in program verification, we propose a method that is incomplete but pragmatic. In reality, most of these proof obligations could be discharged by a small and adequate set of axioms for NIA (associativity, commutativity of non-linear multiplication, distributivity, etc). However, when added to an already large context, such axioms overwhelm SMT solvers and render their use impractical. Our approach aims at making efficient use of these NIA axioms by a built-in treatment in the solver core. Our contributions are as follows:

- An algorithm for non-linear integer arithmetic reasoning (illustrated in Section II); this algorithm relies on the extension and collaboration of the AC(X) framework [13] and interval calculus [8] to handle NIA axioms in a built-in way. AC(X) is instantiated with linear integer arithmetic (LIA) to handle equalities of LIA and associativity and commutativity properties of non linear multiplication. The interval calculus component is used — in addition to standard linear operations over inequalities — to propagate bounds of non-linear terms and to inform the SAT solver about judicious case-splits on finite domains.
- A formalization of this algorithm with a precise description of the cooperation mechanisms in Section III.
- An implementation of this cooperative framework in the SMT solver Alt-Ergo. This implementation uses ground AC-completion and an efficient simplex-based algorithm for LIA interval inference.
- A set of benchmarks generated by program verification tools, to show that our method is competitive.

II. A Collaborative Framework

This section illustrates our collaborative algorithm on the following conjunction of literals, represented as a set for convenience.

$$\left\{ \begin{array}{ll} v * t = 3, & v * w = 5, \\ -(y * y * y) + 3 \cdot w - 5 \cdot t \leq -10, & \\ 0 \leq x \leq 5, & 2 \cdot z * (x/y) + 3 \cdot x = 4, \\ 3 \cdot (x/y) * x \leq 0 & \end{array} \right\}$$

This formula is typical of the kind we want to check for satisfiability. The symbol $*$ denotes non-linear multiplication, the operator $\cdot$ represents linear multiplication (*i.e.* repeated addition) and the symbol $/$ is the Euclidean division. In this example v, w, x, y, z and t are uninterpreted integer constants.

To handle such a formula, our method mainly relies on the collaboration of three components. Figure 1 shows the simplified architecture of this arithmetic framework in Alt-Ergo.

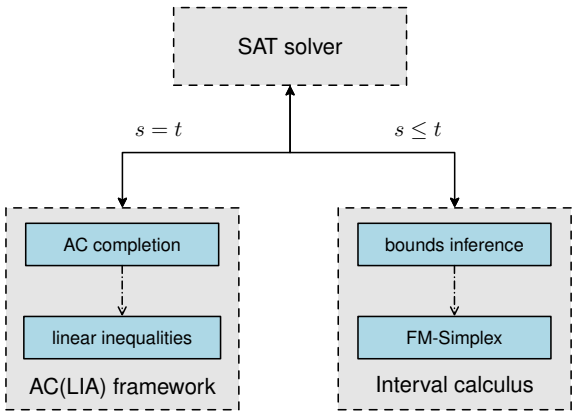

Figure 1: Simplified overview of Alt-Ergo's arithmetic reasoning framework

The first component – on the left hand-side of the figure – is a completion-like algorithm AC(LIA) to reason modulo associativity and commutativity properties of non-linear multiplication, as well as its distributivity over addition. The AC(LIA) framework consists of a modular extension of ground AC completion with a decision procedure that reasons modulo equalities of linear integer arithmetic. This component builds and maintains a convergent term rewriting system modulo equalities of LIA and the AC properties of the $*$ symbol.

The second component – on the right of the figure – is an interval calculus algorithm used to compute bounds of (non-linear) terms. First, the initial NIA problem is relaxed to LIA by abstracting non-linear sub-terms and a simplex-based algorithm (FM-Simplex) is used to infer bounds on the abstracted linear problem. Second, axioms of NIA are internally applied by intervals propagation.

The third module is a SAT solver that dispatches equalities to AC(LIA) and inequalities to interval calculus. The SAT also performs case-split analysis over finite domains (*i.e.* bounded intervals) computed by interval calculus.

The proof of unsatisfiability of the example is carried out as follows in our collaborative framework:

- using the AC properties of the * symbol and a straightforward critical pairs computation, the AC(LIA) procedure deduces that $3 * w = 5 * t$ follows from the first and the second equalities. Since $3 * w$ and $5 * t$ are linear terms, the equality simplifies to $3 \cdot w = 5 \cdot t$;
- using the deduction above, the third equality simplifies to $-(y * y * y) \leq -10$. Then, the interval calculus deduces that $y \geq 3$;
- now, using $y \geq 3$ and $0 \leq x \leq 5$, the interval calculus component refines the bounds of x/y to the integer interval $[0; 1]$. At this point, we have to perform a case-split analysis to conclude;
- if $x/y = 0$, the term $z * (x/y)$ becomes linear and simplifies to 0. Thus, the fifth hypothesis normalizes to $3 \cdot x = 4$. The linear integer arithmetic solver, provided by LIA, says that the equality has no solution in $\mathbb{Z}$;
- when $x/y = 1$, the last hypothesis becomes $3 \cdot x \geq 0$. Together with the fourth assumption, this implies $x = 0$. However, the interval calculus component deduces that this equality contradicts the last case-split $x/y = 1$.

III. Formal Description

In this section, we formally define the collaborative framework described in Figure 1. We start by the SAT solver module, which is a standard CDCL solver modulo theory [33], and show how it integrates the cooperation of the AC(LIA) and interval calculus modules.

A. SAT Module

We assume the usual syntactic and semantic notions of first-order logic. In particular, we denote by $M \models F$ the logical entailment relation between formulas. For convenience, conjunctions are represented by sets of formulas. We also assume a background theory $\mathcal{T}$.

When Mode $=$ **search**:

$$\textsc{Success}\ \frac{M \models F}{\text{return}\ \ \textsc{Sat}} \qquad \textsc{Decide}\ \frac{l \text{ is undefined in } M \qquad l \text{ (or } \neg l) \in F}{M := l :: M}$$

$$\textsc{Propagate}\ \frac{C \vee l \in F \qquad M \models \neg C \qquad l \text{ undef in } M}{M := l_{C \vee l} :: M} \qquad \textsc{Model-Based Case-Split}\ \frac{\{l\} \cup M \ \text{ is } \ \mathcal{T}\text{-sat}}{M := l :: M}$$

$$\mathcal{T}\text{-}\textsc{Propagate}\ \frac{\{l_1, \ldots, l_n, \neg l\} \ \text{ is } \mathcal{T}\text{-unsat} \qquad \{l_1, \ldots, l_n\} \subseteq M \qquad l \text{ undef in } M}{M := l_{\neg l_1 \vee \ldots \neg l_n \vee l} :: M}$$

$$\textsc{Conflict}\ \frac{C \in F \qquad M \models \neg C}{R := C;\ \texttt{Mode} := \text{resolution}} \qquad \mathcal{T}\text{-}\textsc{Conflict}\ \frac{\{l_1, \ldots, l_n\} \subseteq M \qquad \{l_1, \ldots, l_n\} \ \text{ is } \mathcal{T}\text{-unsat}}{R := \neg l_1 \vee \ldots \neg l_n;\ \texttt{Mode} := \text{resolution}}$$

When Mode $=$ **resolution**:

$$\textsc{Fail}\ \frac{R \ \text{ is the empty clause}}{\text{return}\ \ \textsc{Unsat}} \qquad \textsc{Resolve}\ \frac{R = C \vee \neg l \qquad l_{D \vee l} \in M}{R := C \vee D}$$

$$\textsc{Backjump}\ \frac{R = C \vee l \quad M = M_1 :: l' :: M_2 \qquad M_2 \models \neg C \qquad l \text{ is undefined in } M_2}{M := l_{C \vee l} :: M_2;\ \texttt{Mode} := \text{search}}$$

Figure 2: CDCL solver modulo theory

The state of our SAT module is defined by four global variables M, F, R and Mode where

- F is a set of clauses
- M is a partial model, represented by a *stack* of literals; we use the notation $M_1 :: M_2$ for the concatenation of stacks M_1 and M_2 (for convenience, $l :: M$ denotes a stack with l at its top);
- R is a *conflict* clause
- Mode is variable equal to **search** or **resolution**

Literals in M are of two forms: we distinguish between *decision* literals l, and *implied* literals l_C annotated with an *explanation* clause C. When convenient, we treat M as a set of literals (in that case, we ignore subscripts of implied literals).

The SAT algorithm is abstractly defined by the non-deterministic state transition system in Figure 2. Following [30], each transition rule is given in a *guard/action* form. Actions of a rule are enabled only when its guards hold. We distinguish between two kinds of actions: state variable assignments, and **return** statements.

Following standard CDCL solvers, our SAT engine has two distinct behaviors: a *search* mode (Mode = **search**) and a *resolution* mode (Mode = **resolution**). For the sake of simplicity, we split the set of rules according to the content of variable Mode.

When Mode = **search**. The SAT solver terminates by returning SAT if M is a model for F (rule SUCCESS). Rule DECIDE makes a new decision and push the new decision literal l on top of M. Boolean constraint propagation is done by rule PROPAGATE. Rule MODEL-BASED CASE-SPLIT implements a case split similar to the model-based combination described in [16]. It allows a new decision literal l to be added on top of M when there exists a model $\mathcal{M}$ for the theory T coherent with the literals in M. Similarly to PROPAGATE, rule T-PROPAGATE performs constraint propagation at the theory level. Rule CONFLICT detects a conflict at the boolean level, assigns variable R with the conflict clause C and switches to the resolution mode. Similarly, T-CONFLICT detects a conflict at the theory level.

When Mode = **resolution**. The SAT solver terminates by returning UNSAT if R contains the empty clause $\bot$ (rule FAIL). Rule RESOLVE performs a resolution step between the clause in R and the explanation clause of an implied literal in M. Finally, rule BACKJUMP performs non-chronological backtracking and switches back to the search mode.

Note that only three rules, MODEL-BASED CASE-SPLIT, T-PROPAGATE and T-CONFLICT, have an interaction with the theory modules. We describe in the next two sections how the literals in M are handled by the interval calculus and the AC(LIA) modules, and postpone to Section IV the concrete implementation of model-based case-splits.

Configurations of the theory modules are of the form $\langle\ M \mid I \mid R\ \rangle$ where

- M is a set of equations, inequations and disequations between terms in normal form with respect to LIA;
- I is a map from arithmetic terms (affine forms $\Sigma_i \lambda_i t_i$) to disjoint unions of intervals;
- R is a rewriting system, *i.e.* a set of oriented equalities

For deciding if a set of literals M is $\mathcal{T}$-satisfiable, we run the theory modules starting from the initial configuration $\langle\ M \mid \emptyset \mid \emptyset\ \rangle$.

B. AC(LIA) Module

We describe the AC(LIA) module by the set of inference rules of in Figure 3. These rules can be applied in any order. This module assumes given a canonizer and a solver for LIA.

$$\textsc{DBottom}\ \frac{\langle\ M \cup \{\, s \neq s \,\} \mid I \mid R\ \rangle}{\text{return}\ \ \textsc{Unsat}} \qquad \textsc{Bottom}\ \frac{\langle\ M \cup \{\, s = t \,\} \mid I \mid R\ \rangle}{\text{return}\ \ \textsc{Unsat}}\ \texttt{solve}(s,t) = \bot$$

$$\textsc{Orient}\ \frac{\langle\ M \cup \{\, s = t \,\} \mid I \mid R\ \rangle}{\langle\ M \mid I \mid R \cup \texttt{solve}(s,t)\ \rangle}\ \texttt{solve}(s,t) \neq \bot \qquad \textsc{Simplify}\ \frac{\langle\ M \cup \{\, s \bowtie t \,\} \mid I \mid R\ \rangle}{\langle\ M \cup \{\, s' \bowtie t \,\} \mid I \mid R\ \rangle}\ s \rightsquigarrow_R s'$$

$$\textsc{Compose}\ \frac{\langle\ M \mid I \mid R \cup \{\, l \to r \,\}\ \rangle}{\langle\ M \mid I \mid R \cup \{\, l \to r' \,\}\ \rangle}\ r \rightsquigarrow_R r'$$

$$\textsc{Collapse}\ \frac{\langle\ M \mid I \mid R \cup \{\, g \to d,\, l \to r \,\}\ \rangle}{\langle\ M \cup \{\, l' = r \,\} \mid I \mid R \cup \{\, g \to d \,\}\ \rangle}\ \begin{cases} l \rightsquigarrow_{g \to d} l' \\ g \prec l \ \vee\ (g \simeq l \wedge d \prec r) \end{cases}$$

$$\textsc{Deduce}\ \frac{\langle\ M \mid I \mid R\ \rangle}{\langle\ M \cup \texttt{headCP}(R) \mid I \mid R\ \rangle}$$

Figure 3: Inference rules for AC(LIA)

Its canonizer is a function $\texttt{can}_{\text{LIA}}$ that computes a unique normal form for every term such that $s =_{\text{LIA}} t$ iff $\texttt{can}_{\text{LIA}}(s) = \texttt{can}_{\text{LIA}}(t)$. A solver for LIA is a function $\texttt{solve}_{\text{LIA}}$ that, given an equation $s = t$, where s and t are LIA terms, either returns a special value $\bot$ when $s = t$ is inconsistent modulo LIA, or an equivalent substitution.

Our AC(LIA) algorithm is based on the integration of $\texttt{can}_{\text{LIA}}$ and $\texttt{solve}_{\text{LIA}}$ in ground AC-completion.

In order to deal with NIA terms, and in particular to cope with the AC properties of the non-linear multiplication symbol, we adapt the LIA canonizer to go through non-linear symbols. Following the technique described in [29], we define a global canonizer `can` by combining $\texttt{can}_{\text{LIA}}$ with the canonizer for AC defined in [24] and formally proved in [14]. For instance, $\texttt{can}((2 \cdot (x * y) + (y * x))/x)$ gives the term $(3 \cdot (x * y))/x$.

Using the same technique, we define a wrapper `solve` to handle NIA equations by interpreting non-linear terms as black boxes. See [29] for a more thorough definition of canonizers and solvers, and [13] for specific requirements in this setting.

In order to integrate `can` in ground AC completion, we adapt the notion of ground AC-rewriting to cope with canonizers. From rewriting point of view, a canonizer behaves like a convergent rewriting system: it gives an effective way of computing normal forms. Thus, a natural way for integrating `can` in ground AC-completion is to extend normalized rewriting [31] by replacing normalization with canonization.

Canonized rewriting A term s rewrites to a canonical term t by the rule $l \to r$, denoted by $s \rightsquigarrow_{l \to r} t$, if and only if s rewrites to t' by $l \to r$ modulo AC and $\texttt{can}(t') = t$.

In order to ensure termination of AC(LIA), we assume (following the AC(X) framework [13]) the global canonizer `can` and the wrapper `solve` are compatible with a given total ground AC-reduction ordering $\preceq$.

The first rule DBOTTOM returns UNSAT when M contains trivial inconsistent disequations. Similarly, rule BOTTOM is used to detect trivial inconsistent equations by calling `solve`.

Equations are turned into rewriting rules by ORIENT which adapts the orientation mechanism of ground AC-completion. Given an equation $s = t$, ORIENT adds the substitution returned by $\texttt{solve}(s = t)$ to R. This rule only applies when `solve` returns a solution for an equation.

All remaining rules are similar to those of ground AC-completion, except that we replace the AC-rewriting relation by our canonized rewriting relation $\rightsquigarrow$. In SIMPLIFY, the rewriting system R is used to reduce either side of literals $s \bowtie t$, where $\bowtie$ stands for $=, \neq$ or $\leq$. Similarly, COMPOSE reduces right hand sides of rewriting rules. Given a rule $l \to r$, COLLAPSE either reduces l at an inner position, or replaces l by a term smaller than r. In both cases, the reduction of l to l' may influence the orientation of the rule $l' \to r$ which is added to M as an equation in order to be re-oriented. Finally, DEDUCE adds equational consequences of rewriting rules to M. For instance, if R contains two rules of the form $a*b \to s$ and $a * c \to t$, then the term $a * (b * c)$ can either be reduced modulo AC to $s*c$ or to the term $t*b$. The equation $s*c = t*b$, called *critical pair*, is thus necessary for ensuring convergence of R. Critical pairs of a set of rewriting rules are computed by the following function (where a^μ stands for the maximal term (w.r.t. size) enjoying the assertion):

$$\texttt{headCP}(R) = \left\{ b * r' = b' * r \;\middle|\; \begin{array}{l} l \to r \in R,\quad l' \to r' \in R \\ \exists\, a^\mu.\ \ l =_{AC} a^\mu * b \\ \qquad \wedge\ l' =_{AC} a^\mu * b' \end{array} \right\}$$

C. Interval Calculus Module

Notations. We write $[\![a; b]\!]$ for the integer interval bounded by $a, b \in \mathbb{Z}$. We also write $]-\infty; a]\!]$ for $\{x \in \mathbb{Z} \mid x \leq a\}$ and $[\![a; +\infty[$ for $\{x \in \mathbb{Z} \mid x \geq a\}$. We use a simple bracket notation $\langle . \rangle$ in place of $[\![.]\!]$, $]\!.]\!]$, $[\![.[$ or $]\!.[$. Interval multiplication by $k \in \mathbb{Z}$ is $k \cdot \langle a; b\rangle = \langle min(k \cdot a, k \cdot b); max(k \cdot a, k \cdot b)\rangle$. Interval translation by $k \in \mathbb{Z}$ is $\langle a; b\rangle + k = \langle a + k; b + k\rangle$.

Our inference calculus is described by the set of inference rules in Figure 4.

The first rule INCONSISTENT-BOUNDS returns UNSAT if the map I contains a binding $t \mapsto \langle c_1; c_2\rangle$ where the interval $\langle c_1; c_2\rangle$ is reduced to the empty set. Rule IMPLIED-EQUALITY re-injects bindings of the form $t \mapsto [\![c; c]\!]$ as equalities $t = c$

$$\textsc{Inconsistent-bounds}\ \frac{\langle\ M \mid I \cup \{t \to \langle c_1; c_2\rangle \cup D\} \mid R\ \rangle}{\bot}\ c_1 > c_2$$

$$\textsc{Implied-equality}\ \frac{\langle\ M \mid I \cup \{t \to [\![c; c]\!]\} \mid R\ \rangle}{\langle\ \{t = c\} \cup M \mid I \mid R\ \rangle}\ t \not\rightsquigarrow_R^* c \text{ and } (t = c) \notin M$$

$$\textsc{Lia-Bounds}\ \frac{\langle\ M \mid I \mid R\ \rangle}{\langle\ M \mid \texttt{Fm-Simplex}(M) \cup I \mid R\ \rangle}\ \texttt{Fm-Simplex}(M) \cup I \neq I$$

$$\textsc{Nia-saturation}\ \frac{\langle\ M \mid I \mid R\ \rangle}{\langle\ M \mid \texttt{apply\_nia}(I) \mid R\ \rangle} \qquad \textsc{Normalize}\ \frac{\langle\ M \mid I \mid R\ \rangle}{\langle\ M \mid \texttt{norm}(I, R) \mid R\ \rangle}\ \texttt{norm}(I, R) \neq I$$

Figure 4: Inference rules for the interval calculus

in M for future consideration by AC(LIA). In rule LIA-BOUNDS, bindings of I are populated by bounds computed by the `Fm-Simplex` algorithm (described below). Union of bindings is defined in a standard way: two bindings $t \mapsto D$ and $t \mapsto D'$ with the same key t are merged into $t \mapsto D \cap D'$. Intervals in I are then refined by applying of a set of non-linear interval arithmetic axioms by rule NIA-SATURATION (a non-exhaustive list of these axioms is given below). In rule NORMALIZE, terms p that are keys of the map I are normalized so that $a \cdot p$ has the same normal form as p for all $a \in \mathbb{Z}$. At the same time `norm` normalizes the monomials of p with respect to the rewriting system R to ensure that intervals are maintained modulo equality.

The function `Fm-Simplex` takes as input a set of literals M. We note $Iq = \bigcup_i L_i \leq 0$ the subset of M that are inequations. `Fm-Simplex` returns refined intervals for the initial affine forms L_i of Iq. Non-linear terms of Iq are abstracted as simple variables and I is initialized with terms and sub-terms of M (to $]-\infty; +\infty[$). To compute these intervals `Fm-Simplex` uses an efficient Simplex-based implementation [8]. This algorithm attempts to compute constant positive linear combinations $\sum \lambda_i L_i$ (where $\lambda_i \in \mathbb{Q}^+$) that simulates particular projections of the Fourier-Motzkin [19], [28] algorithm. These combinations are then used to infer bounds as shown in below.

Example Consider the following set of affine forms:

$$C_1 : \begin{cases} L_1 = 2x + y, & L_2 = -2x + 3y - 5, \\ L_3 = x + z + 1, & L_4 = x + 5y + z, \\ L_5 = -x - 4y + 3, & L_6 = 3x - 2y + 2 \end{cases}$$

Eliminating z from C_1 is immediate since it only appears positively:

$$C_2 : \begin{cases} L_1 = 2x + y, & L_2 = -2x + 3y - 5, \\ L_5 = -x - 4y + 3, & L_6 = 3x - 2y + 2 \end{cases}$$

We eliminate the variable x and compute the set C_3 below using the combinations: $L_7 = L_1 + L_2$, $L_8 = L_1 + 2L_5$, $L_9 = 2L_6 + 3L_2$, $L_{10} = L_6 + 3L_5$

$$C_3 : \begin{cases} L_7 = 4y - 5, & L_8 = -7y + 6, \\ L_9 = 5y - 11, & L_{10} = -14y + 11 \end{cases}$$

Finally, the variable y is in turn eliminated thanks to the following combinations: $L_{11} = 7L_7 + 4L_8$, $L_{12} = 7L_7 + 2L_{10}$, $L_{13} = 7L_9 + 5L_8$, $L_{14} = 14L_9 + 5L_{10}$

The iterative process terminates and returns the set

$$C_4 : \begin{cases} L_{11} = -11, & L_{12} = -13, \\ L_{13} = -47 & L_{14} = -99 \end{cases}$$

Moreover, unfolding the equalities yields

$$\begin{cases} -11 = L_{11} = 7L_7 + 4L_8 = \cdots = 11L_1 + 7L_2 + 8L_5 \\ -13 = L_{12} = 7L_7 + 2L_{10} = \cdots = 7L_1 + 7L_2 + 6L_5 + 2L_6 \\ -47 = L_{13} = 7L_9 + 5L_8 = \cdots = 5L_1 + 21L_2 + 10L_5 + 14L_6 \\ -99 = L_{14} = 14L_9 + 5L_{10} = \cdots = 42L_2 + 15L_5 + 33L_6 \end{cases}$$

Using the linear combination $11L_1 + 7L_2 + 8L_5 = -11$, we can make the deductions $-1 \leq L_1$, $-\frac{11}{7} \leq L_2$ and $-\frac{11}{8} \leq L_3$ in the rationals. Furthermore, these deductions are refined as follows in the integers: $-1 \leq L_1$, $\lceil -\frac{11}{7} \rceil = -1 \leq L_2$ and $\lceil -\frac{11}{8} \rceil = -1 \leq L_3$.

More formally, `Fm-Simplex` tries to compute one particular constant positive linear combination by solving auxiliary rational optimization problems of the form:

$$\begin{array}{ll} \text{maximize} & \sum_i b_i\, \lambda_i \\ \text{subject to} & \bigwedge_j \sum_i a_{i,j} \cdot \lambda_i = 0 \ \wedge \\ & \sum_i \lambda_i > 0 \ \wedge\ \bigwedge_i \lambda_i \geq 0 \\ & \text{where } L_i = \sum_j a_{i,j} \cdot t_j + b_i \end{array}$$

such that:

- if *unsatisfiable*, there is no constant positive linear combination of the original inequalities – in this case no bounds are inferred so Iq is satisfiable;
- if the optimization problem is unbounded, then Iq is inconsistent;
- otherwise, there exists a negative maximum and a positive linear combination from which bounds can be refined (as shown above).

The function `apply_nia` saturates the map I with axioms of non-linear arithmetic over intervals [23]. For non-linear multiplication, ten axioms are integrated in the solver, shown

below:

$$\forall a,b,c,d \in \mathbb{Z} \cup \{-\infty,+\infty\}, x \in \langle a;b\rangle, y \in \langle c;d\rangle.$$
$$\begin{cases} 0 \le a \le b \ \wedge\ 0 \le c \le d & \Longrightarrow \quad x*y \in \langle a*c; b*d\rangle \\ 0 \le a \le b \ \wedge\ 0 < c < d & \Longrightarrow \quad x*y \in \langle b*c; b*d\rangle \\ 0 \le a \le b \ \wedge\ c \le d \le 0 & \Longrightarrow \quad x*y \in \langle b*c; a*d\rangle \\ a < 0 < b \ \wedge\ 0 \le c \le d & \Longrightarrow \quad x*y \in \langle a*d; b*d\rangle \\ a < 0 < b \ \wedge\ c < 0 < d & \Longrightarrow \\ \qquad\qquad x*y \in \langle min(a*d, b*c); max(a*c, b*d)\rangle & \\ a < 0 < b \ \wedge\ c \le d \le 0 & \Longrightarrow \quad x*y \in \langle b*c; a*c\rangle \\ a \le b \le 0 \ \wedge\ 0 \le c \le d & \Longrightarrow \quad x*y \in \langle a*d; b*c\rangle \\ a \le b \le 0 \ \wedge\ c < 0 < d & \Longrightarrow \quad x*y \in \langle a*d; a*c\rangle \\ a \le b \le 0 \ \wedge\ c \le d \le 0 & \Longrightarrow \quad x*y \in \langle b*d; a*c\rangle \\ a = b = 0 \vee c = d = 0 & \Longrightarrow \quad x*y \in [\![0;0]\!] \end{cases}$$

Our implementation also handles non-linear Euclidean division of polynomials of the form $\lambda P(\bar{x})/P(\bar{x})$ with the axiom:

$$\forall a,b \in \mathbb{Z} \cup \{-\infty,+\infty\}, P(\bar{x}) \in \langle a;b\rangle, \lambda \in \mathbb{Z}.$$
$$0 \notin \langle a;b\rangle \implies \lambda P(\bar{x})/P(\bar{x}) \in [\![\lambda;\lambda]\!]$$

The more general form $P(\bar{x})/Q(\bar{x})$ is handled when $Q(\bar{x}) > 0$ or $Q(\bar{x}) < 0$. For instance, from $x \in [\![0;5]\!]$ and $y \in [\![3;+\infty[$ we deduce $x/y \in [\![0;1]\!]$.

Intervals are also propagated from non-linear terms to their sub-terms when applicable. For example, what are the bounds of x, knowing those of x^n? Our main concern with this technique is to guarantee that our interval arithmetic computes bounds that remain correct by over-approximating the intervals. This is particularly apropos in the case of the root of an interval where results may become irrational numbers.

$$\forall a,b \in \mathbb{Z} \cup \{-\infty,+\infty\}, x^n \in \langle a;b\rangle.$$
$$\begin{cases} n \text{ is odd} & \Longrightarrow\ x \in \langle UA_{\mathbb{Z}}(\sqrt[n]{a})\ ;\ OA_{\mathbb{Z}}(\sqrt[n]{b})\rangle \\ n \text{ is even and } 0 \le a \le b & \Longrightarrow \\ & \qquad x \in \langle -OA_{\mathbb{Z}}(\sqrt[n]{b})\ ;\ OA_{\mathbb{Z}}(\sqrt[n]{b})\rangle \end{cases}$$

where

$$UA_{\mathbb{Z}}(\sqrt[n]{x}) = \lceil UA_{\mathbb{Q}}(\sqrt[n]{x})\rceil \text{ and } OA_{\mathbb{Z}}(\sqrt[n]{x}) = \lfloor OA_{\mathbb{Q}}(\sqrt[n]{x})\rfloor$$

are accurate (under and over) approximations of $\sqrt[n]{x}$ in $\mathbb{Z}$ computed as follows.

Let $A_{\mathbb{Q}}(\sqrt[n]{x})$ be an accurate approximation in $\mathbb{Q}$ of $\sqrt[n]{x}$ and $OA_{\mathbb{Q}}(\sqrt[n]{x})$ (resp. $UA_{\mathbb{Q}}(\sqrt[n]{x})$) be an over-approximation (resp. under-approximation) of $\sqrt[n]{x}$. We can safely deduce that:

if $A_{\mathbb{Q}}(\sqrt[n]{x})^n > x$ then
$OA_{\mathbb{Q}}(\sqrt[n]{x}) = A_{\mathbb{Q}}(\sqrt[n]{x})$ and $UA_{\mathbb{Q}}(\sqrt[n]{x}) = x/A_{\mathbb{Q}}(\sqrt[n]{x})^{n-1}$

if $A_{\mathbb{Q}}(\sqrt[n]{x})^n < x$ then
$UA_{\mathbb{Q}}(\sqrt[n]{x}) = A_{\mathbb{Q}}(\sqrt[n]{x})$ and $OA_{\mathbb{Q}}(\sqrt[n]{x}) = x/A_{\mathbb{Q}}(\sqrt[n]{x})^{n-1}$

if $A_{\mathbb{Q}}(\sqrt[n]{x})^n = x$ then $OA_{\mathbb{Q}}(\sqrt[n]{x}) = UA_{\mathbb{Q}}(\sqrt[n]{x}) = A_{\mathbb{Q}}(\sqrt[n]{x})$

For example, from $-y^3 \in\ ]-\infty;-10]\!]$ we can deduce that $y \in [\![3;+\infty[$.

Function `norm` normalizes keys of the map I with the rewriting system R. Every binding $p \mapsto D$ of I is replaced with a normalized binding to ensure that identities between terms differing only from a multiplicative factor will be discovered and that their intervals will be merged. An affine form p has to be reduced with the rules of R, *i.e.* $p \rightsquigarrow_R p' + c$ where p' is an affine form $\Sigma_i a_i \cdot t_i$. p' is then normalized by a multiplicative factor $k = \frac{a_1}{|a_1|} \cdot \frac{ppcm(\ldots,a_i,\ldots)}{pgcd(\ldots,a_i,\ldots)}$ which guarantees that coefficients $k \cdot a_i$ remain integer numbers and that the sign of p' is also normalized with respect to its first coefficient. $p \mapsto D$ is replaced by (and merged in I) the equivalent binding $k \cdot p' \mapsto k \cdot (D - c)$.

Additionally, if `norm` encounters a key of the form p/q, the axiom:

$$\forall q,p \in \mathbb{Z}.\ \exists \text{ a unique } k \in \mathbb{Z}.\ q = (q/p) * p + k \wedge 0 \le k \le p$$

is instantiated by adding the following new bindings to I, knowing that this unique k only depends on p and q and is noted $p\%q$:

$$q - (q/p) * p - (p\%q) \mapsto [\![0;0]\!]$$
$$(p\%q) \mapsto [\![0;+\infty[$$
$$(p\%q) - q \mapsto]-\infty;0]\!]$$

IV. Implementation

We integrated our collaborative framework in the Alt-Ergo SMT-solver. The architecture of the current implementation is very close to description of Figure 1. We describe in this section some implementation details of the rules given in Section III. In particular, we explain how the premises of rules Model-Based Case-Split, $\mathcal{T}$-propagate and $\mathcal{T}$-conflict (in Figure 2) are implemented. Furthermore, we give our deterministic strategy for applying rules of Figures 3 and 4.

An important feature from an implementation view point, is that our theory modules should be incremental and backtrackable. Indeed, the SAT module constructs its partial model M in an incremental way, and context of successive calls to theory modules only differ in just a few literals. Additionally, when the SAT backjumps, theories must recover their previous states. In Alt-Ergo, theories modules are implemented with persistent data structures (sets, maps, etc.) thus backtrackability is obtained for free.

For now, all components are incremental and backtrackable except for `Fm-Simplex`. Each call to `Fm-Simplex` processes the literals of M from scratch. To circumvent this issue, we employ memoization techniques to reuse previously computed results at the expense of a larger memory footprint.

Concerning the implementation of rule Model-Based Case-Split, the difficulty is to discover the case-split literal l. Alt-Ergo finds such a literal by looking for a binding $t \mapsto [\![c_1;c_2]\!]$ in I such that $c_2 - c_1 \ge 1$. In this case, the case-split literal is $t = c_1$. As a heuristic, priority is given to bindings with the smallest intervals.

In the rule $\mathcal{T}$-conflict, the difficulty is to find the subset $\{l_1,\ldots,l_n\}$ of M that is $\mathcal{T}$-unsatisfiable. For that, we have implemented an explanation mechanism that consists in remembering the set of literals that were involved in each deduction step (bound inference, adding a rewriting rule, etc.) of the theory modules. Note that a special attention should be

given to the implementation of this explanation mechanism; if literals are missing from $\{l_1, \ldots, l_n\}$, the SAT solver may backjump to the wrong decision level.

The additional difficulty of $\mathcal{T}$-PROPAGATE is to find the implied literal l. We restrict the possible literals to the ones (or their negation) that appear in F. The key point of the implementation is that each time the SAT solver assumes a new literal, we rapidly identify all literals impacted by this addition. This is implemented thanks to an indirection table that dispatches literals according to their sub-terms. We extract all implied literals from the set returned by a look-up in this table.

We used the following strategy to process an equation using the inference rules of AC(LIA):

SIMPLIFY*
(BOTTOM | (ORIENT (COMPOSE COLLAPSE DEDUCE)*))

This means that the equation is first simplified as much as possible. Then, if it is not proven to be trivially unsolvable, it is solved. Each resulting rule is added to the rewriting system and then used to "compose" and "collapse" the other rules of R. Finally, critical pairs are computed and added to M.

Solving an equation can also have an impact in the interval calculus module. In this case, the following strategy is used to normalize the map I with respect to R and to call bounds inference rules.

NORMALIZE LIA-BOUNDS NIA-SATURATION

Similarly, processing an inequality is done as follows:

SIMPLIFY* LIA-BOUNDS NIA-SATURATION

Notice that, INCONSISTENT-BOUNDS (*resp.* IMPLIED-EQUALITY) is applied as soon as an interval becomes inconsistent (*resp.* reduces to a point).

V. EXPERIMENTAL RESULTS

In this section, we evaluate our collaborative framework on a collection of verification conditions issued from program verification. The aims of our evaluation is to show that:

- although incomplete, our approach allows us to prove formulas requiring a simple non-linear integer arithmetic reasoning;
- our extension does not slow down the Alt-Ergo SMT-solver when non-linear integer arithmetic reasoning is not needed.

For the experiments, we have used the current svn revision of Alt-Ergo and a modified version of it where non-linear integer arithmetic is deactivated. We also used the latest versions of some state-of-the-art SMT solvers including CVC3 (version 2.4.1), CVC4 (version 1.2) and Z3 (version 4.3.1). Our test suite is composed of three benchmarks which already contain some auxiliary NIA axioms to help provers that don't support built-in NIA reasoning:

- the first one is made of 3431 formulas generated from the SPARK Hi-Lite toolset[1]. These verification conditions were only available in Alt-Ergo's native input language;
- the second one contains 80 difficult verification conditions generated from the gallery of programs of Why, version 2. Formulas in this benchmark were only available in Alt-Ergo's native input language;
- the third benchmark is composed of 1920 verification conditions issued from Why3's gallery of verified programs[2]. These formulas where generated in Alt-Ergo's native input language, the SMTLIB-2 language and CVC3's native input language.

All measures were obtained on a 64-bit machine with a quad-core Intel Xeon processor at 3.2 GHz and 24 GB of memory. Provers were given a time limit of 60 seconds and a memory limit of 2 GB for each verification condition.

	unsat	time	unk.	time
alt-ergo	**2285**	1017 s	621	5329 s
ae no-nia	2241	1241 s	704	5709 s

Figure 5: Benchmark issued from the SPARK Hi-Lite toolset.

	unsat	time	unk.	time
alt-ergo	**57**	45 s	18	176 s
ae no-nia	34	44 s	40	180 s

Figure 6: Formulas issued from Why's gallery of programs.

	unsat	time	unk.	time
alt-ergo	**1842**	**465 s**	19	45 s
ae no-nia	1830	456 s	30	85 s
z3	1528	392 s	1	0.1 s
cvc3	1767	382 s	34	290 s
cvc4	1664	619 s	2	0.1 s

Figure 7: Benchmark issued from Why3's gallery of programs.

The results of our experiments are reported in Figures 5, 6 and 7. The first column of each table shows the number of formulas *solved* by each prover. The second one reports the corresponding accumulated time. The third and the fourth columns report the number of formulas for which the provers returned *unknown* and the corresponding time, respectively.

From these figures, we remark that Alt-Ergo proves more formulas when non-linear integer arithmetic is activated. Moreover, we were surprised to notice on Figure 5 that Alt-Ergo with NIA is 18% faster than Alt-Ergo without NIA. We also notice from figures 6 and 7 that the overhead of our non-linear arithmetic extension is negligible.

These results can be explained because:

- Alt-Ergo has more built-in NIA axioms than what is given in these benchmarks;

[1] Available on http://libre.adacore.com/tools/spark-gpl-edition/

[2] Available on http://toccata.lri.fr/gallery/why3.en.html

- some of these auxiliary NIA axioms are likely to be ill-suited for the instantiation mechanism;
- other auxiliary NIA axioms, *e.g.* associativity and commutativty, will glut the solver with plenty of useless instances.

In a second step, we tried to evaluate our approach on the UF-NIA benchmark of SMT-LIB [5]. However, we noticed that formulas in this benchmark require a non-trivial preprocessing of LET-IN and IF-THEN-ELSE high-level constructs. Unfortunately, such a capability is not provided by Alt-Ergo for the moment.

VI. Conclusion and Future Works

In this paper, we have presented a collaborative approach of procedures for reasoning in the linear and non-linear fragment of integer arithmetic. We have implemented this framework in the Alt-Ergo theorem prover and the first experiments show that this method is promising. Further improvements on the combination of interval arithmetic with the rest of the framework include the incorporation of other NIA axioms and the extension of the matching algorithm — used when applying these axioms — modulo AC properties.

References

[1] Z. S. Andraus and et al. Automatic abstraction and verification of verilog models, 2004.

[2] D. Babic and M. Musuvathi. Modular arithmetic decision procedure. *Microsoft Research Redmond, Tech. Rep. TR-2005-114*, 2005.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

[4] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krstić, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke. C.: The smt-lib standard: Version 2.0. Technical report, 2010.

[6] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[7] P. Baudin, F. Bobot, R. Bonichon, L. Correnson, P. Cuoq, Z. Dargaye, J.-C. Filliâtre, P. Herrmann, F. Kirchner, M. Lemerre, C. Marché, B. Monate, Y. Moy, A. Pacalet, V. Prévosto, J. Signoles, and B. Yakobowski. Frama-c. http://frama-c.com.

[8] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, A. Mahboubi, A. Mebsout, and G. Melquiond. A simplex-based extension of fourier-motzkin for solving linear integer arithmetic. In *Automated Reasoning*, pages 67–81. Springer, 2012.

[9] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, et al. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[10] M. Bozzano, R. Bruttomesso, A. Cimatti, A. Franzén, Z. Hanna, Z. Khasidashvili, A. Palti, and R. Sebastiani. Encoding rtl constructs for mathsat: a preliminary report. *Electron. Notes Theor. Comput. Sci.*, 144(2):3–14, Jan. 2006.

[11] C. W. Brown. Qepcad b: A program for computing with semi-algebraic sets using cads. *SIGSAM BULLETIN*, 37:97–108, 2003.

[12] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *SIGSAM Bull.*, 10(1):10–12, Feb. 1976.

[13] S. Conchon, E. Contejean, and M. Iguernelala. Canonized rewriting and ground ac completion modulo shostak theories: Design and implementation. *arXiv preprint arXiv:1207.3262*, 2012.

[14] E. Contejean. A certified AC matching algorithm. In V. van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, volume 3091, pages 70–84, Aachen, Germany, June 2004.

[15] E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[16] L. M. de Moura and N. Bjørner. Model-based theory combination. *Electr. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[17] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary*, volume 4963, pages 337–340. Springer, 2008.

[18] B. Dutertre and L. D. Moura. The yices smt solver. Technical report, SRI, 2006.

[19] J.-B. J. Fourier. Reported in: Analyse des travaux de l'Académie Royale des Sciences, pendant l'année 1824, Partie mathématique, Histoire de l'Académie Royale des Sciences de l'Institut de France. (7), 1827.

[20] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[21] M. K. Ganai. Efficient decision procedure for bounded integer non-linear operations using smt (\ mathcal {LIA}). In *Hardware and Software: Verification and Testing*, pages 68–83. Springer, 2009.

[22] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[23] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48:1038–1068, September 2001.

[24] J.-M. Hullot. Associative commutative pattern matching. In *Proc. 6th IJCAI (Vol. I), Tokyo*, pages 406–412, Aug. 1979.

[25] D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2012.

[26] J. Kanig, E. Schonberg, and C. Dross. Hi-lite: the convergence of compiler technology and program verification. In B. Brosgol, J. Boleng, and S. T. Taft, editors, *HILT*, pages 27–34. ACM, 2012.

[27] D. Kapur. Using gröbner bases to reason about geometry problems. *J. Symb. Comput.*, 2(4):399–408, 1986.

[28] D. Kroening and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[29] S. Krstić and S. Conchon. Canonization for disjoint unions of theories. *Information and Computation*, 199(1-2):87–106, May 2005.

[30] S. Krstic and A. Goel. Architecting solvers for sat modulo theories: Nelson-oppen with dpll. In *Frontiers of Combining Systems, Liverpool, UK, September 10-12, 2007, Proceedings*, volume 4720, pages 1–27. Springer, 2007.

[31] C. Marché. Normalized rewriting: an alternative to rewriting modulo a set of equations. 21(3):253–288, 1996.

[32] Y. V. Matiyasevich. Enumerable sets are diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

[33] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Abstract DPLL and abstract DPLL modulo theories. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'04), Montevideo, Uruguay*, volume 3452 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[34] G. O. Passmore. Combined decision procedures for nonlinear arithmetics, real and complex. 2011.

[35] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid sat-based decision procedure for separation logic with uninterpreted functions. In *In Proc. DAC'03*, pages 425–430, 2003.

[36] N. Smart. *The Algorithmic Resolution of Diophantine Equations: A Computational Cookbook*. London Mathematical Society Student Texts. Cambridge University Press, 1998.

[37] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[38] A. C. Ward and W. Seering. An approach to computational aids for mechanical design. In *Proceedings of the International Conference on Engineering Design*, 1981.

[39] V. Weispfenning. *A New Approach to Quantifier Elimination for Real Algebra*. Fakultät für Mathematik und Informatik: MIP. Fak. für Math. und Informatik, Univ. Passau, 1993.