A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo
Sylvain Conchon, Mohamed Iguernelala, Alain Mebsout

To cite this version:
Sylvain Conchon, Mohamed Iguernelala, Alain Mebsout. A Collaborative Framework for Non-Linear Integer Arithmetic Reasoning in Alt-Ergo. 2013. hal-00924646

HAL Id: hal-00924646
https://hal.archives-ouvertes.fr/hal-00924646
Submitted on 7 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 SPARX Hi-Lite [24] 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:

\[
\forall n, p, r \in \mathbb{Z}. \quad n \geq 0 \land p = n \land r = \frac{n + 1}{2} \Rightarrow r = \frac{p + n}{2}
\]

```
/*@ 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 dio-phantine (in)equations, a restricted form of non-linear integer constraints [36]. While this problem has also been shown to be undecidable [12], 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.

\[
\begin{align*}
&v \cdot t = 3, \quad v \cdot w = 5, \\
&-(y \cdot y + y) + 3 \cdot w - 5 \cdot t \leq -10, \\
&0 \leq x \leq 5, \quad 2 \cdot z \cdot (x/y) + 3 \cdot x = 4, \\
&3 \cdot (x/y) \cdot x \leq 0
\end{align*}
\]

This formula is typical of the kind we want to check for satisfiability. The symbol \( \cdot \) denotes non-linear multiplication, the operator \( / \) 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.

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 \( \ast \) 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 \( \ast \) symbol and a straightforward critical pairs computation, the AC(LIA) procedure deduces that \( 3 \ast w = 5 \ast t \) follows from the first and the second equalities. Since \( 3 \ast w \) and \( 5 \ast t \) are linear terms, the equality simplifies to \( 3 \cdot w = 5 \cdot t \); 

• using the deduction above, the third equality simplifies to \( -(y \cdot y + y) \leq -10 \). 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 \( z \cdot (x/y) \) 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 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 \( T \).
The state of our SAT module is defined by four global variables $M$, $F$, $R$ and $\text{Mode}$ 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
- $\text{Mode}$ 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 ($\text{Mode} =$ search) and a resolution mode ($\text{Mode} =$ resolution). For the sake of simplicity, we split the set of rules according to the content of variable $\text{Mode}$.

When $\text{Mode} =$ 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 $M$ 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 $\text{Mode} =$ 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 $T$-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 in Figure 3. These rules can be applied in any order. This module assumes given a canonizer and a solver for LIA.
Its canonizer is a function \( \text{can}_{\text{LIA}} \) that computes a unique normal form for every term such that \( s =_{\text{LIA}} t \) iff \( \text{can}_{\text{LIA}}(s) = \text{can}_{\text{LIA}}(t) \). A solver for LIA is a function \( \text{solve}_{\text{LIA}} \) 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 \( \text{can}_{\text{LIA}} \) and \( \text{solve}_{\text{LIA}} \) 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 \( \text{can} \) by combining \( \text{can}_{\text{LIA}} \) with the canonizer for AC defined in [24] and formally proved in [14]. For instance, \( \text{can} \left( 2 \cdot (x + y) + (y \cdot x) \right) / x \) gives the term \( 3 \cdot (x + y) / x \).

Using the same technique, we define a wrapper \( \text{solve} \) 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 \( \text{can} \) 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 \( \text{can} \) in ground AC-completion is to extend normalized rewriting [31] by replacing normalization with canonization.

**Canonical rewriting** A term \( s \) rewrites to a canonical term \( t \) by the rule \( l \rightarrow r \), denoted by \( s \overset{\mu}{\rightarrow}_{r} t \), if and only if \( s \) rewrites to \( t' \) by \( l \rightarrow r \) modulo AC and \( \text{can}(t') = t \).

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

The first rule \( \text{DBottom} \) returns \( \text{UNSAT} \) when \( M \) contains trivial inconsistent disequations. Similarly, rule \( \text{Bottom} \) is used to detect trivial inconsistent equations by calling \( \text{solve} \).

Equations are turned into rewriting rules by \( \text{ORIENT} \) which adapts the orientation mechanism of ground AC-completion.

Given an equation \( s = t \), \( \text{ORIENT} \) adds the substitution returned by \( \text{solve}(s = t) \) to \( R \). This rule only applies when \( \text{solve} \) 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 \( \rightarrow_{\mu} \). In \( \text{SIMPLIFY} \), the rewriting system \( R \) is used to reduce either side of literals \( s \overset{\mu}{\rightarrow}_{r} t \), where \( \infty \) stands for \( =, \neq \) or \( \leq \). Similarly, \( \text{COMPOSE} \) reduces right hand sides of rewriting rules. Given a rule \( l \rightarrow r \), \( \text{COLLAPSE} \) 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 \( l' \rightarrow r \) which is added to \( M \) as an equation in order to be re-oriented. Finally, \( \text{DEDUCE} \) adds equational consequences of rewriting rules to \( M \). For instance, if \( R \) contains two rules of the form \( a \cdot b \rightarrow s \) and \( a \cdot c \rightarrow t \), then the term \( a \cdot (b \cdot c) \) can either be reduced modulo AC to \( s \cdot c \) or to the term \( t \cdot b \). The equation \( s \cdot c = t \cdot b \), 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 \( a^\mu \) stands for the maximal term w.r.t. size) enjoying the assertion:

\[
\text{headCP}(R) = \left\{ \begin{array}{l}
\{ b \cdot r' = b' \cdot r \} & l \rightarrow r \in R, \ l' \rightarrow r' \in R \\
\exists a^\mu. \ l =_{AC} a^\mu \cdot b \\
\wedge \ l' =_{AC} a^\mu \cdot b'
\end{array} \right\}
\]

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 \(\langle r \rangle\) in place of \([r; r]\). 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 \( \text{INCONSISTENT-BOUNDS} \) returns \( \text{UNSAT} \) 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 \( \text{IMPLIED-EQUALITY} \) re-injects bindings of the form \( t \mapsto \langle c; c \rangle \) as equalities \( t = c \).
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

\[
\begin{align*}
-11 &= L_{11} = L_7 + 4L_8 = \cdots = 11L_1 + 7L_2 + 8L_5 \\
-13 &= L_{12} = L_7 + 2L_10 = \cdots = 7L_1 + 7L_2 + 6L_5 + 2L_6 \\
-47 &= L_{13} = 7L_9 + 5L_8 = \cdots = 5L_1 + 21L_2 + 10L_5 + 14L_6 \\
-99 &= L_{14} = 14L_9 + 5L_{10} = \cdots = 42L_2 + 15L_5 + 33L_6
\end{align*}
\]

Using the linear combination \(11L_1 + 7L_2 + 8L_5 = -11\), we can make the deductions \(-1 \leq L_1, -\frac{11}{2} \leq L_2\) and \(-\frac{11}{8} \leq L_3\) in the rationals. Furthermore, these deductions are refined as follows in the integers: \(-1 \leq L_1, \left[-\frac{11}{2}\right] = -1 \leq L_2\) and \(\left[-\frac{11}{8}\right] = -1 \leq L_3\).

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

\[
\text{maximize} \quad \sum_i b_i \lambda_i \\
\text{subject to} \quad \lambda_j \sum_i a_{i,j} \cdot \lambda_i = 0 \land \lambda_j \lambda_i > 0 \land \land_i \lambda_i \geq 0
\]

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).

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 function \texttt{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:

\[
\forall a, b, c, d \in \mathbb{Z} \cup \{-\infty, +\infty\}, \ x \in (a; b), \ y \in (c; d).
\]

\[
\begin{cases}
0 \leq a \leq b \land 0 \leq c \leq d \implies x \ast y \in (a \ast c; b \ast d) \\
0 \leq a \leq b \land 0 < c < d \implies x \ast y \in (b; c \ast b \ast d) \\
0 \leq b \land c \leq d \leq 0 \implies x \ast y \in (b; c\ast a \ast d) \\
a < 0 < b \land 0 \leq c \leq d \implies x \ast y \in (a \ast d; b; d) \\
a < 0 < b \land c < 0 < d \implies x \ast y \in (a \ast d; b; c) \\
\end{cases}
\]

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

\[
\forall a, b, c, d \in \mathbb{Z} \cup \{-\infty, +\infty\}, \ P(\vec{x}) \in (a; b), \lambda \in \mathbb{Z},
\]

\[
0 \not\in (a; b) \implies \lambda P(\vec{x})/P(\vec{x}) \in [\lambda; \lambda]
\]

The more general form \(P(\vec{x})/Q(\vec{x})\) is handled when \(Q(\vec{x}) > 0\) or \(Q(\vec{x}) < 0\). 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 sub-terms when applicable. For example, what are the sub-terms when applicable.

Additionally, if \(\text{norm}\) encounters a key of the form \(p/q\), the axiom:

\[
\forall q, p \in \mathbb{Z}. \exists a \text{ unique } k \in \mathbb{Z}. \ q = (q/p) \ast p + k \land 0 \leq k \leq p
\]

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

\[
q - (q/p) \ast p - (p\%q) \rightarrow [0; 0] \quad (p\%q) \rightarrow [0; +\infty] \\
(p\%q) - q \rightarrow ]-\infty; 0]\]

### 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, T-PROPAGATE and 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 \(\text{Fm-Simplex}\). Each call to \(\text{Fm-Simplex}\) 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 \(t \rightarrow [a_1; c_2]\) in \(I\) such that \(c_2 - c_1 \geq 1\). 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 T-CONFLICT, the difficulty is to find the subset \(\{l_1, \ldots, l_n\}\) of \(M\) that is 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 \(T\)-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):

\[
\text{SIMPLIFY}^\ast
\]

\((\text{BOTTOM} \mid (\text{ORIENT} \ (\text{COMPOSE} \ \text{COLLAPSE} \ \text{DEDUCE})^*))\)

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.

\[
\text{NORMALIZE} \ \text{LIA-BOUNDS} \ \text{NIA-SATURATION}
\]

Similarly, processing an inequality is done as follows:

\[
\text{SIMPLIFY}^\ast \ \text{LIA-BOUNDS} \ \text{NIA-SATURATION}
\]

Notice that, \text{INCONSISTENT-BOUNDS} (resp. \text{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.

<table>
<thead>
<tr>
<th></th>
<th>unsat</th>
<th>time</th>
<th>unk.</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt-ergo</td>
<td>2285</td>
<td>1017 s</td>
<td>621</td>
<td>5329 s</td>
</tr>
<tr>
<td>ae no-nia</td>
<td>2241</td>
<td>1241 s</td>
<td>704</td>
<td>5709 s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>unsat</th>
<th>time</th>
<th>unk.</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt-ergo</td>
<td>57</td>
<td>45 s</td>
<td>18</td>
<td>176 s</td>
</tr>
<tr>
<td>ae no-nia</td>
<td>34</td>
<td>44 s</td>
<td>40</td>
<td>180 s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>unsat</th>
<th>time</th>
<th>unk.</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>alt-ergo</td>
<td>1842</td>
<td>465 s</td>
<td>19</td>
<td>45 s</td>
</tr>
<tr>
<td>ae no-nia</td>
<td>1830</td>
<td>456 s</td>
<td>30</td>
<td>85 s</td>
</tr>
<tr>
<td>z3</td>
<td>1528</td>
<td>392 s</td>
<td>1</td>
<td>0.1 s</td>
</tr>
<tr>
<td>cvc3</td>
<td>1767</td>
<td>382 s</td>
<td>34</td>
<td>290 s</td>
</tr>
<tr>
<td>cvc4</td>
<td>1664</td>
<td>619 s</td>
<td>2</td>
<td>0.1 s</td>
</tr>
</tbody>
</table>

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 5 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;

---


• some of these auxiliary NIA axioms are likely to be ill-suited for the instantiation mechanism;
• other auxiliary NIA axioms, e.g. associativity and commutativity, 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 internal 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