Journal on Satisfiability, Boolean Modeling and Computation 8 (2012) 1-27

A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic

Alberto Griggio[*] griggio@fbk.eu

Fondazione Bruno Kessler - IRST
Via Sommarive 18, Povo, Trento
Italy

Abstract

We present a detailed description of a theory solver for Linear Integer Arithmetic ($\mathcal{LA}(\mathbb{Z})$) in a lazy SMT context. Rather than focusing on a single technique that guarantees theoretical completeness, the solver makes extensive use of layering and heuristics for combining different techniques in order to achieve good performance in practice. The viability of our approach is demonstrated by an empirical evaluation on a wide range of benchmarks, showing significant performance improvements over current state-of-the-art solvers.

Keywords: *SMT, linear integer arithmetic, decision procedures*

Submitted April 2010; revised February 2011; published January 2012

1. Introduction

Due to its many important applications, Linear Arithmetic is one of the most well-studied theories in SMT. In particular, current state-of-the-art SMT solvers are very effective in dealing with quantifier-free formulas over Linear *Rational* Arithmetic ($\mathcal{LA}(\mathbb{Q})$), incorporating very efficient decision procedures for it [13, 16]. However, support for Linear *Integer* Arithmetic ($\mathcal{LA}(\mathbb{Z})$) is not as mature yet. Although several SMT solvers support $\mathcal{LA}(\mathbb{Z})$, experiments show that they are not very good at handling $\mathcal{LA}(\mathbb{Z})$-formulas which require a significant amount of integer reasoning (that is, formulas for which a $\mathcal{LA}(\mathbb{Q})$-model with non-integer values can be easily found), and that they can easily get lost in searching for a solution of small and apparently-simple problems. In our opinion, as recently observed also in [12], this indicates that the theory solvers for $\mathcal{LA}(\mathbb{Z})$ currently used (e.g. [14, 21]), although theoretically complete, are not very robust in practice.

In this paper, we present a new theory solver for $\mathcal{LA}(\mathbb{Z})$ which explicitly focuses on achieving good performance on practical examples. The key feature of our solver is an extensive use of *layering* and *heuristics* for combining different known techniques, in order to exploit the strengths and overcome the limitations of each of them. Such approach was inspired by the work on (Mixed) Integer Linear Programming solvers, in which heuristics play a crucial role for performance [1]. We give a detailed description of the main techniques that we have implemented, and discuss issues and choices for an efficient integration of the

[*] Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

solver in a lazy SMT system based on the DPLL($\mathcal{T}$) architecture [23, 20]. Finally, we demonstrate the viability of our approach by evaluating our implementation (as part of our new SMT solver MathSAT5) on a wide range of benchmarks, showing significant performance improvements over current state-of-the-art solvers.

The rest of the paper is organized as follows. In §2 we introduce the necessary background concepts and terminology used in lazy SMT. In §3 we describe the general architecture of our $\mathcal{LA}(\mathbb{Z})$-solver and the relationships among its different modules. Details about the two main modules are then given in §4 and §5. In §6 we describe the experimental evaluation, and in §7 we conclude. The discussion of related work is distributed in the technical sections (§4 and §5).

2. Lazy Satisfiability Modulo Theories

Our setting is standard first order logic. A 0-ary function symbol is called a *constant*. A *term* is a first-order term built out of function symbols and variables. If $t_1, \ldots, t_n$ are terms and p is a predicate symbol, then $p(t_1, \ldots, t_n)$ is an *atom*. In this paper, we shall only deal with atoms that are either 0-arity predicates (i.e. Boolean constants), or *linear equations and inequalities* $\sum_i a_i x_i + c \bowtie 0$, where $\bowtie \in \{=, \leq\}$, c and the a_i's are rational numbers and the x_i's are uninterpreted integer constants. A *formula* ϕ is built in the usual way out of the universal and existential quantifiers, Boolean connectives, and atoms. A *literal* is either an atom or its negation. We call a formula *quantifier-free* if it does not contain quantifiers, and *ground* if it does not contain free variables.

We also assume the usual first-order notions of interpretation, satisfiability, validity, logical consequence, and theory, as given, e.g., in [15]. We write $\Gamma \models \phi$ to denote that the formula ϕ is a logical consequence of the (possibly infinite) set Γ of formulas. A *first-order theory*, $\mathcal{T}$, is a set of first-order sentences. A structure $\mathcal{A}$ is a model of a theory $\mathcal{T}$ if $\mathcal{A}$ satisfies every sentence in $\mathcal{T}$. A formula is *satisfiable in* $\mathcal{T}$ (or $\mathcal{T}$*-satisfiable*) if it is satisfiable in a model of $\mathcal{T}$. (We sometimes use the word "$\mathcal{T}$-formula" for a ground formula when we are interested in determining its $\mathcal{T}$-satisfiability.)

Given a first-order formula ϕ, the *propositional abstraction* of ϕ is a propositional formula ψ obtained from ϕ by replacing each theory atom in ϕ with a fresh Boolean constant. We assume to have a mapping $\mathcal{T}2\mathcal{B}$ ("theory to Boolean") from theory atoms to fresh Boolean constants and its inverse $\mathcal{B}2\mathcal{T}$ ("Boolean to theory") which can be used to obtain the propositional abstraction ψ from a formula ϕ and vice versa.

In what follows, we might denote conjunctions of literals $l_1 \wedge \ldots \wedge l_n$ as sets $\{l_1, \ldots, l_n\}$ and vice versa. If $\eta \equiv \{l_1, \ldots, l_n\}$, we might write $\neg\eta$ to mean $\neg l_1 \vee \ldots \vee \neg l_n$. Moreover, following the terminology of the SAT and SMT communities, we shall refer to predicates of arity zero as *propositional variables*, and to uninterpreted constants as *theory variables*.

Given a first-order theory $\mathcal{T}$ for which the (ground) satisfiability problem is decidable, we call a *theory solver for* $\mathcal{T}$, $\mathcal{T}$-solver, any tool able to decide the satisfiability in $\mathcal{T}$ of sets/conjunctions of ground atomic formulas and their negations — *theory literals* or $\mathcal{T}$-*literals* — in the language of $\mathcal{T}$. If the input set of $\mathcal{T}$-literals μ is $\mathcal{T}$-unsatisfiable, then a typical $\mathcal{T}$-solver not only returns `unsat`, but it also returns the subset η of $\mathcal{T}$-literals in μ which was found $\mathcal{T}$-unsatisfiable. (η is hereafter called a *theory conflict set*, and $\neg\eta$ a *theory conflict clause*.) If μ is $\mathcal{T}$-satisfiable, then $\mathcal{T}$-solver not only returns `sat`, but it may also be able to discover one (or more) deductions in the form $\{l_1, \ldots, l_n\} \models_{\mathcal{T}} l$, s.t. $\{l_1, \ldots, l_n\} \subseteq \mu$ and l is an unassigned $\mathcal{T}$-literal. If so, we call $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ a *theory-deduction clause.* Importantly, notice that both theory-conflict clauses and theory-deduction clauses are valid in $\mathcal{T}$. We call them *theory lemmas* or *$\mathcal{T}$-lemmas.*

Satisfiability Modulo (the) Theory $\mathcal{T}$ — SMT($\mathcal{T}$) — is the problem of deciding the satisfiability of *Boolean combinations* of propositional atoms and theory atoms. We call an *SMT($\mathcal{T}$) tool* any tool able to decide SMT($\mathcal{T}$). Notice that, unlike a $\mathcal{T}$-solver, an SMT($\mathcal{T}$) tool must handle also Boolean connectives.

Hereafter we adopt the following terminology and notation. The symbols φ, ψ denote $\mathcal{T}$-formulas, and μ, η denote sets of $\mathcal{T}$-literals; φ^p, ψ^p denote propositional formulas, μ^p, η^p denote sets of propositional literals (i.e., truth assignments) and we often use them as synonyms for the propositional abstraction of φ, ψ, μ, and η respectively, and vice versa (e.g., φ^p denotes $\mathcal{T}2\mathcal{B}(\varphi)$, μ denotes $\mathcal{B}2\mathcal{T}(\mu^p)$). If $\mathcal{T}2\mathcal{B}(\varphi) \models \bot$, then we say that φ is *propositionally unsatisfiable.*

2.1 The Online Lazy SMT Schema

The currently most popular approach for solving the SMT($\mathcal{T}$) problem is the so-called *"lazy"* approach [23, 4], also frequently called "DPLL($\mathcal{T}$)" [20].

The lazy approach works by combining a propositional SAT solver based on the DPLL algorithm [10] with a $\mathcal{T}$-solver. Essentially, DPLL is used as an enumerator of truth assignments μ_i^p propositionally satisfying the propositional abstraction φ^p of the input formula φ, and the $\mathcal{T}$-solver is used for checking the $\mathcal{T}$-satisfiability of each $\mu_i \stackrel{\text{def}}{=} \mathcal{B}2\mathcal{T}(\mu_i^p)$: if the current μ_i is $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-satisfiable; otherwise, if none of the μ_i's are $\mathcal{T}$-satisfiable, then φ is $\mathcal{T}$-unsatisfiable.

Figure 1 represents the schema of a modern lazy SMT solver based on a DPLL engine (see e.g. [25]). It is an abstraction of the algorithm implemented in most state-of-the-art lazy SMT solvers, including BarceLogic [5], CVC3 [3], OpenSMT [9], Yices2 [13], Z3 [11] and MathSAT5.

The input φ and μ are a $\mathcal{T}$-formula and a reference to an (initially empty) set of $\mathcal{T}$-literals respectively. The DPLL solver embedded in $\mathcal{T}$-DPLL reasons on and updates φ^p and μ^p, and $\mathcal{T}$-DPLL maintains some data structure encoding the bijective mapping $\mathcal{T}2\mathcal{B}/\mathcal{B}2\mathcal{T}$ on atoms.

$\mathcal{T}$-preprocess simplifies φ into a simpler formula while preserving its $\mathcal{T}$-satisfiability. If this process produces some conflict, then $\mathcal{T}$-DPLL returns unsat. $\mathcal{T}$-preprocess may combine most or all the Boolean preprocessing steps available from SAT literature with theory-dependent rewriting steps on the $\mathcal{T}$-literals of φ. This step involves also the conversion of φ to CNF, if required.

$\mathcal{T}$-decide-next-branch selects some literal l^p and adds it to μ^p. It plays the same role as the standard literal selection heuristic decide-next-branch in DPLL [25], but it may take into consideration also the semantics in $\mathcal{T}$ of the literals to select.

$\mathcal{T}$-deduce, in its simplest version, behaves similarly to deduce in DPLL [25], i.e. it iteratively performs Boolean Constraint Propagation (BCP). This step is repeated until one of the following events occur:

```
SatValue $\mathcal{T}$-DPLL ($\mathcal{T}$-formula $\varphi$, reference $\mathcal{T}$-assignment $\mu$)
if $\mathcal{T}$-preprocess ($\varphi$, $\mu$) == conflict then
    return unsat
end if
$\varphi^p = \mathcal{T}2\mathcal{B}(\varphi)$
$\mu^p = \mathcal{T}2\mathcal{B}(\mu)$
loop
    $\mathcal{T}$-decide-next-branch ($\varphi^p$, $\mu^p$)
    loop
        status = $\mathcal{T}$-deduce ($\varphi^p$, $\mu^p$)
        if status == sat then
            $\mu = \mathcal{B}2\mathcal{T}(\mu^p)$
            return sat
        else if status == conflict then
            $\langle$blevel, $\eta\rangle$ = $\mathcal{T}$-analyze-conflict ($\varphi^p$, $\mu^p$)
            if blevel $< 0$ then
                return unsat
            else
                $\mathcal{T}$-backtrack (blevel, $\varphi^p$, $\mu^p$, $\eta^p$)
            end if
        else
            break
        end if
    end loop
end loop
```

Figure 1. An online schema of $\mathcal{T}$-DPLL based on modern DPLL.

(i) μ^p propositionally violates φ^p ($\mu^p \wedge \varphi^p \models \bot$). If so, $\mathcal{T}$-deduce behaves like deduce in DPLL, returning conflict.

(ii) μ^p satisfies φ^p ($\mu^p \models \varphi^p$). If so, $\mathcal{T}$-deduce invokes $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}(\mu^p)$: if $\mathcal{T}$-solver returns sat, then $\mathcal{T}$-deduce returns sat; otherwise, $\mathcal{T}$-deduce returns conflict.

(iii) no more literals can be deduced. If so, $\mathcal{T}$-deduce returns unknown.

A slightly more elaborated version of $\mathcal{T}$-deduce can invoke $\mathcal{T}$-solver on $\mathcal{B}2\mathcal{T}(\mu^p)$ also if μ^p does not yet satisfy φ^p: if $\mathcal{T}$-solver returns unsat, then $\mathcal{T}$-deduce returns conflict. (This enhancement is called *Early Pruning*, EP.)

Moreover, during EP calls, if $\mathcal{T}$-solver is able to perform deductions in the form $\eta \models_{\mathcal{T}} l$ s.t. $\eta \subseteq \mu$ and $l^p \overset{\text{def}}{=} \mathcal{T}2\mathcal{B}(l)$ is an unassigned literal in φ^p, then $\mathcal{T}$-deduce can append l^p to μ^p and propagate it. (This enhancement is called *$\mathcal{T}$-propagation*.)

$\mathcal{T}$-analyze-conflict is an extension of analyze-conflict of DPLL [25], in which conflict analysis is performed either on the clause falsified during BCP in $\mathcal{T}$-deduce (case (i) above), or on the propositional abstraction $\eta'^p \overset{\text{def}}{=} \mathcal{T}2\mathcal{B}(\eta')$ of the $\mathcal{T}$-conflict set η' produced by $\mathcal{T}$-solver (case (ii) above). In both cases, $\mathcal{T}$-analyze-conflict produces a conflict set η^p and the corresponding value blevel of the decision level where to backtrack.

$\mathcal{T}$-backtrack behaves analogously to backtrack in DPLL [25]: once the conflict set η^p and blevel have been computed, it adds the clause $\neg\eta^p$ to φ^p, either temporarily or permanently, and backtracks up to blevel. (These features are called $\mathcal{T}$-learning and $\mathcal{T}$-backjumping.)

An important enhancement of $\mathcal{T}$-deduce is to use a technique called *layering* [7, 8], which consists of using a collection of $\mathcal{T}$-solvers $S_1, \ldots, S_N$ organized in a *layered hierarchy* of increasing expressivity and complexity. Each solver S_i is able to decide a theory $\mathcal{T}_i$ which is a subtheory of $\mathcal{T}_{i+1}$, and which is less expensive to handle than $\mathcal{T}_{i+1}$. The solver S_N is the only one that can decide the full theory $\mathcal{T}$. If the solver S_i detects an inconsistency, then there is no need of invoking the more expensive solvers $S_{i+1}, \ldots, S_N$, and unsat can be returned immediately.

Another important extension of $\mathcal{T}$-deduce is a technique called "splitting on-demand" [2]. With splitting on-demand, $\mathcal{T}$-solvers are not required to be always able to decide the $\mathcal{T}$-consistency of the current set of constraints μ: rather, they might sometimes return unknown, together with a list of *new $\mathcal{T}$-lemmas containing new $\mathcal{T}$-atoms*, which will be then taken into account in the DPLL search, by branching on the new atoms and performing BCP and conflict detection on the new lemmas. The idea of splitting on-demand is that of exploiting DPLL for performing disjunctive reasoning, instead of handling it inside the $\mathcal{T}$-solvers, whenever this is needed for checking the $\mathcal{T}$-consistency of μ. This not only simplifies the implementation of $\mathcal{T}$-solvers, but it also allows to take advantage for free of all the advanced techniques (like conflict-driven backjumping and learning) for search-space pruning implemented in modern DPLL engines.

Finally, we remark that during EP calls in $\mathcal{T}$-deduce the $\mathcal{T}$-solvers are allowed to be *imprecise* in detecting conflicts, in the sense that they are allowed to return sat even when the current truth assignment μ is $\mathcal{T}$-inconsistent, as long as precision is recovered in non-EP calls (i.e. when $\mu^p \models \varphi^p$) [23]. This makes it possible to implement a further enhancement to $\mathcal{T}$-deduce, called *Weak Early Pruning*, in which during EP calls the $\mathcal{T}$-solvers use an *approximate* but cheaper consistency check algorithm, in order to limit the overhead of frequent EP calls. Weak EP is particularly effective on "hard" theories, including $\mathcal{LA}(\mathbb{Z})$.

3. General Architecture of the $\mathcal{LA}(\mathbb{Z})$-solver

Figure 2 shows an outline of the general architecture of our $\mathcal{T}$-solver for $\mathcal{LA}(\mathbb{Z})$ (hereafter, simply $\mathcal{LA}(\mathbb{Z})$-solver).

The solver is organized as a layered hierarchy of submodules, with cheaper (but less powerful) ones invoked earlier and more often. The general strategy used for checking the consistency of a set of $\mathcal{LA}(\mathbb{Z})$-constraints is as follows.

First, the rational relaxation of the problem is checked, using a Simplex-based $\mathcal{LA}(\mathbb{Q})$-solver similar to that described in [13]. If no conflict is detected, the model returned by the $\mathcal{LA}(\mathbb{Q})$-solver is examined to check whether all integer variables are assigned to an integer value. If this happens, the $\mathcal{LA}(\mathbb{Q})$-model is also a $\mathcal{LA}(\mathbb{Z})$-model, and the solver can return sat.

Otherwise, the specialized module for handling linear Diophantine equations is invoked. This module is similar to the first part of the Omega test described in [21]: it takes all the equations in the input problem, and tries to eliminate them by computing a parametric solution of the system and then substituting each variable in the inequalities with its para-

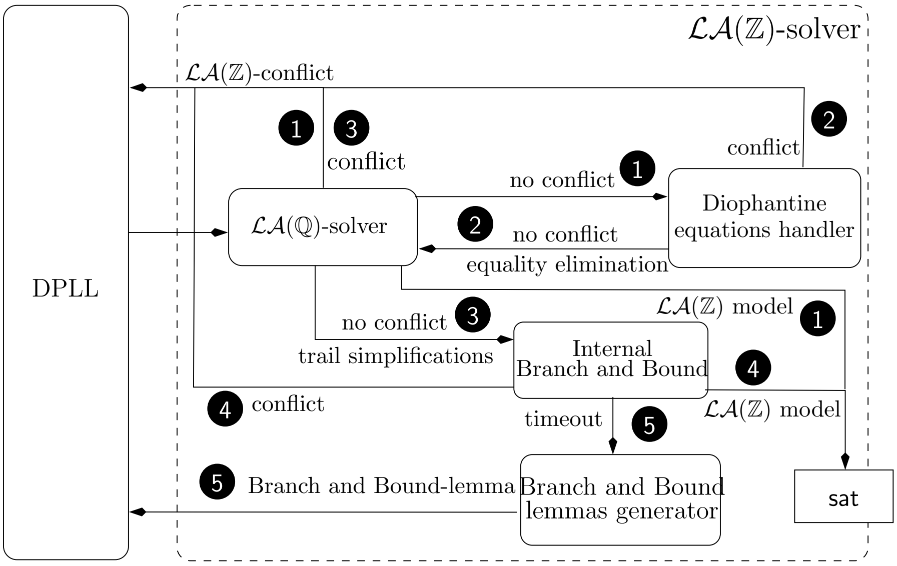

$\mathcal{LA}(\mathbb{Z})$-solver
DPLL
$\mathcal{LA}(\mathbb{Z})$-conflict
1
3
conflict
2
conflict
no conflict
1
Diophantine equations handler
$\mathcal{LA}(\mathbb{Q})$-solver
2
no conflict
equality elimination
$\mathcal{LA}(\mathbb{Z})$ model
1
no conflict
3
trail simplifications
Internal Branch and Bound
4
4
conflict
$\mathcal{LA}(\mathbb{Z})$ model
timeout
5
5
Branch and Bound-lemma
Branch and Bound lemmas generator
sat

Figure 2. Architecture of the $\mathcal{LA}(\mathbb{Z})$-solver.

metric expression. If the system of equations is infeasible in itself, this module is also able to detect the inconsistency.

Otherwise, the inequalities obtained by substituting the variables with their parametric expressions are normalized, tightened and then sent to the $\mathcal{LA}(\mathbb{Q})$-solver, in order to check the $\mathcal{LA}(\mathbb{Q})$-consistency of the new set of constraints.

If no conflict is detected, the branch and bound module is invoked, which tries to find a $\mathcal{LA}(\mathbb{Z})$-solution via branch and bound [22]. This module is itself divided into two submodules operating in sequence. First, the “internal” branch and bound module is activated, which performs case splits directly within the $\mathcal{LA}(\mathbb{Z})$-solver. The internal search is performed only for a bounded (and small) number of branches, after which the “external” branch and bound module is called. This works in cooperation with the DPLL engine, using the splitting on-demand approach of [2]. Splitting on-demand is also used for handling disequalities, by generating a lemma $(t \neq 0) \rightarrow (t+1 \leq 0) \vee (-t+1 \leq 0)$ for each disequality $(t \neq 0)$ seen by the $\mathcal{LA}(\mathbb{Z})$-solver.

4. The Diophantine Equations Handler

The module for handling systems of $\mathcal{LA}(\mathbb{Z})$ equations (commonly called *Diophantine equations*) implements a procedure that closely resembles the equality elimination step of the Omega test [21].

Given a system $E \stackrel{\text{def}}{=} \{\sum_{i=1}^{n} a_{ji}x_i + c_j = 0\}_{j=1}^{m}$ of m equations over n variables, it tries to solve it by performing a sequence of variable elimination steps using the procedure described in Algorithm 1. The algorithm runs in polynomial time [21], and can be easily made incremental.

Algorithm 1: Solving a system of linear Diophantine equations

Input: a system of Diophantine equations E.

Output: a parametric solution S for E, or unsat if E is inconsistent.

1. Let $F = E, S = \emptyset, \hat{e}_h = \mathsf{null}$.

2. If F is empty, the system is consistent; return sat with S as a solution.

3. Rewrite all equations $e_h \overset{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ in F such that the GCD g of $a_{h1}, \ldots, a_{hn}, c_h$ is greater than 1 into $e'_h \overset{\text{def}}{=} \sum_i \frac{a_{hi}}{g}x_i + \frac{c_h}{g} = 0$.

4. If there exists an equation $e_h \overset{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ in F such that the GCD of the a_{hi}'s does not divide c_h, then F is inconsistent (see, e.g., [21]); return unsat.

5. Otherwise, if $\hat{e}_h$ is null, pick an equation $e_h \overset{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ from F, and set $\hat{e}_h = e_h$.

6. Let a_{hk} be the non-zero coefficient with the smallest absolute value in $\hat{e}_h$.

7. If $|a_{hk}| = 1$, then $\hat{e}_h$ can be rewritten as

   $$-x_k + \sum_{i \neq k} -\text{sign}(a_{hk})a_{hi}x_i - \text{sign}(a_{hk})c_h = 0, \tag{1}$$

   where $\text{sign}(a_{hk}) \overset{\text{def}}{=} \dfrac{a_{hk}}{|a_{hk}|}$. Then, remove $\hat{e}_h$ from F, add it to S, replace x_k with $\sum_{i \neq k} -\text{sign}(a_{hk})a_{hi}x_i - \text{sign}(a_{hk})c_h$ in all the other equations of F, and set $\hat{e}_h = \mathsf{null}$.

8. If $|a_{hk}| > 1$, then rewrite $\hat{e}_h$ as

   $$\begin{aligned} a_{hk}x_k + \sum_{i \neq k}(a_{hk}a^q_{hi} + a^r_{hi})x_i + (a_{hk}c^q_h + c^r_h) \; &= 0 \;\equiv \\ a_{hk} \cdot (x_k + \sum_{i \neq k} a^q_{hi}x_i + c^q_h) + (\sum_{i \neq k} a^r_{hi}x_i + c^r_h) &= 0. \end{aligned}$$

   where a^q_{hi} and a^r_{hi} are respectively the quotient and the remainder of the division of a_{hi} by a_{hk} (and similarly for c^q_h and c^r_h). Create a fresh variable x_t, and add to S the equation

   $$-x_k + \sum_{i \neq k} -a^q_{hi}x_i - c^q_h + x_t = 0.$$

   Then, replace x_k with $\sum_{i \neq k} -a^q_{hi}x_i - c^q_h + x_t$ in all the equations of F.

9. Go to Step 2.

Theorem 1 *Algorithm 1 always terminates. Moreover, it returns* unsat *if and only if the input system of Diophantine equations is inconsistent.*

Proof. (Sketch) For correctness, we can observe that:

(i) At every iteration of the loop 2–8, the initial system E is equisatisfiable with the system $F \cup S$.

(ii) S is always consistent, since all its equations are of the form

$$e_j \stackrel{\text{def}}{=} -x_j + \sum_{i \neq j} a_{ij} x_i + c_j = 0, \tag{2}$$

where x_j does not occur in any equation that was added to S after e_j (and therefore it can be easily put in triangular form).

Termination can be established by observing that, after the substitution of x_k with $\sum_{i \neq k} -a^q_{hi} x_i - c^q_h + x_t$ performed in Step 8, the equation $\hat{e}_h$ selected in Step 5 becomes

$$a_{hk} x_t + \sum_{i \neq k} a^r_{hi} x_i + c^r_h. \tag{3}$$

Since (i) the a^r_{hi}'s are the remainders of the division of the a_{hi}'s by a_{hk}, and (ii) the coefficient a_{hk} was chosen to be the minimum in absolute value in $\hat{e}_h$, then each $|a^r_{hi}|$ is strictly smaller than the corresponding $|a_{hi}|$. Therefore, after a finite number of applications of Step 8, the equation $\hat{e}_h$ will contain a variable whose coefficient has an absolute value of 1, and therefore it will be eliminated from F by an application of Step 7 [21]. □

Example 1 *Consider the following system of Diophantine equations*

$$E \stackrel{\textit{def}}{=} \begin{cases} e_1 \stackrel{\textit{def}}{=} 3x_1 + 3x_2 + 14x_3 - 7 = 0 \\ e_2 \stackrel{\textit{def}}{=} 7x_1 + 12x_2 + 31x_3 - 17 = 0 \end{cases}$$

In order to prove its unsatisfiability, a run of Algorithm 1 can proceed as follows:

1. *e_1 is processed. Since there are no variables with coefficient 1 or -1, Step 8 is applied. x_1 is selected, e_1 is rewritten as*

$$3(x_1 + x_2 + 4x_3 - 2) + (2x_3 - 1) = 0,$$

a fresh variable x_4 is created, the equation

$$-x_1 - x_2 - 4x_3 + 2 + x_4 = 0$$

is added to S, and x_1 is substituted with $-x_2 - 4x_3 + 2 + x_4$ in all the equations in F, thus obtaining:

$$S = \{ \; -x_1 - x_2 - 4x_3 + 2 + x_4 = 0$$

$$F = \begin{cases} e'_1 \stackrel{\textit{def}}{=} 3x_4 + 2x_3 - 1 = 0 \\ e'_2 \stackrel{\textit{def}}{=} 5x_2 + 3x_3 + 7x_4 - 3 = 0 \end{cases}$$

2. *e_1' is processed. As before, Step 8 is applied, this time selecting x_3, since it is the variable with the smallest coefficient in absolute value. Then, e_1' is rewritten as*

$$2(x_3 + x_4) + (x_4 - 1) = 0,$$

a fresh variable x_5 is created, the equation

$$-x_3 - x_4 + x_5 = 0$$

is added to S, and x_3 is substituted with $-x_4 + x_5$ in all the equations in F, thus obtaining:

$$S = \begin{cases} -x_1 - x_2 - 4x_3 + 2 + x_4 = 0 \\ -x_3 - x_4 + x_5 = 0 \end{cases}$$

$$F = \begin{cases} e_1'' \overset{def}{=} 2x_5 + x_4 - 1 = 0 \\ e_2'' \overset{def}{=} 5x_2 + 4x_4 + 3x_5 - 3 = 0 \end{cases}$$

3. *e_1'' is processed. This time, since x_4 has coefficient 1, Step 7 is applied, e_1'' is moved to S and x_4 is substituted with $-2x_5 + 1$ in e_2'', thus obtaining:*

$$S = \begin{cases} -x_1 - x_2 - 4x_3 + 2 + x_4 = 0 \\ -x_3 - x_4 + x_5 = 0 \\ -x_4 - 2x_5 + 1 = 0 \end{cases}$$

$$F = \left\{ e_2''' \overset{def}{=} 5x_2 - 5x_5 + 1 = 0 \right.$$

4. *Since the GCD of the coefficients of the variables in e_2''' does not divide the constant value of e_2''', the equation in inconsistent, so the algorithm returns* unsat*.*

◇

If Algorithm 1 returns unsat, the $\mathcal{LA}(\mathbb{Z})$-solver can return unsat. If it returns sat, instead, S can be used to eliminate all the equalities from the problem, using each equation e_j of S as a substitution $x_j \mapsto \sum_{i \neq j} a_{ij} x_i + c_j$.

This elimination might make it possible to *tighten* some of the new inequalities generated. Given an inequality $\sum_i a_i x_i + c \leq 0$ such that the GCD g of the a_i's does not divide the constant c, a tightening step [21] consists in rewriting it into $\sum_i \frac{a_i}{g} x_i + \lceil \frac{c}{g} \rceil \leq 0$. Tightening is important because it might allow the $\mathcal{LA}(\mathbb{Q})$-solver to detect more conflicts, as shown in the following example.

Example 2 *Consider the following sets of $\mathcal{LA}(\mathbb{Z})$-constraints:*

$$E \overset{def}{=} \begin{cases} 2x_1 - 5x_3 = 0 \\ x_2 - 3x_4 = 0 \end{cases} \qquad I \overset{def}{=} \begin{cases} -2x_1 - x_2 - x_3 + 7 \leq 0 \\ 2x_1 + x_2 + x_3 - 8 \leq 0 \end{cases}$$

$E \cup I$ is satisfiable over the rationals, but unsatisfiable over the integers. Therefore, the $\mathcal{LA}(\mathbb{Q})$-solver alone can not detect the inconsistency. Thus, E is given to Algorithm 1, which returns the following solution:

$$S = \begin{cases} -x_1 + 2x_3 + x_5 = 0 \\ -x_2 + 3x_4 = 0 \\ -x_3 + 2x_5 = 0, \end{cases}$$

where x_5 is a fresh variable. Using S, we can eliminate the equalities by substituting x_1, x_2 and x_3 into the inequalities in I, thus obtaining:

$$I' = \begin{cases} -3x_4 - 12x_5 + 7 \leq 0 \\ 3x_4 + 12x_5 - 8 \leq 0 \end{cases}$$

On the integers, the two inequalities in I can be tightened *by dividing the constant by the GCD of the coefficients, and then taking the ceiling of the result:*

$$I'' = \begin{cases} -\frac{3}{3}x_4 - \frac{12}{3}x_5 + \lceil\frac{7}{3}\rceil \leq 0 & \textit{which becomes} \quad -x_4 - 4x_5 + 3 \leq 0 \\ \frac{3}{3}x_4 + \frac{12}{3}x_5 + \lceil-\frac{8}{3}\rceil \leq 0 & \textit{which becomes} \quad x_4 + 4x_5 - 2 \leq 0 \end{cases}$$

After this, the $\mathcal{LA}(\mathbb{Q})$-solver can immediately detect the inconsistency of I''. ◇

From the point of view of the implementation, the communication between the Diophantine equation handler and the $\mathcal{LA}(\mathbb{Q})$-solver is made possible by the fact that, differently from what is described in [13], our $\mathcal{LA}(\mathbb{Q})$-solver does not assume that only *elementary bounds* of the form $(x - c \leq 0)$ are asserted and retracted during search, but rather it supports the addition and deletion of arbitrary constraints. [1.]

4.1 Generating explanations for conflicts and substitutions

An important capability of the Diophantine equations handler is its ability to produce *explanations* for conflicts, expressed in terms of a subset of the input equations. This is needed not only when an inconsistency is detected by Algorithm 1 directly (in order to return to DPLL the corresponding $\mathcal{LA}(\mathbb{Z})$-conflict clause), but also when an inconsistency is detected by the $\mathcal{LA}(\mathbb{Q})$-solver after the elimination of the equalities and the tightening of the inequalities. In this case, in fact, the explanation returned by the $\mathcal{LA}(\mathbb{Q})$-solver can not be used directly to generate a conflict clause to give back to DPLL, since it might contain some inequalities that were generated by the equality elimination and tightening step. When this happens, each of these inequalities must be replaced with the original inequality and the set of equations that were used to obtain it. Therefore, the Diophantine equations handler must be able to identify the set of input equations that were used for generating a substitution in the returned solution S.

[1.] The distinction between tableau equations and elementary bounds introduced in [13] is still used, but such transformation is performed internally in the $\mathcal{LA}(\mathbb{Q})$-solver rather than at preprocessing time [17]. As a matter of fact, the original motivation for this decision was to ease the generation of proofs of unsatisfiability; however, having such more general interface turns out to be convenient (from an implementation point of view) also in the present context.

In order to describe how explanations are generated and to prove that the procedure is correct, we introduce an abstract transition system whose inference rules mirror the basic steps performed by Algorithm 1. We then show how the states and the transitions of such system can be annotated with additional information used to produce explanations. Finally, we give a proof of the correctness of the generated explanations.

The basic steps performed by Algorithm 1 can be described as manipulations of a set of equations E according to the following rules:

Scaling of an equation

$$\begin{array}{r} E \cup \{\sum_i a_i x_i + c = 0\} \rightarrow \\ E \cup \{(\sum_i a_i x_i + c = 0), (\sum_i \frac{a_i}{g} x_i + \frac{c}{g} = 0)\} \\ \text{if } g = GCD(a_i, \ldots, a_n, c) \text{ and } g > 1. \end{array} \tag{4}$$

Combination of two equations

$$\begin{array}{r} E \cup \{(\sum_i a_{1i} x_i + c_1 = 0), (\sum_i a_{2i} x_i + c_2 = 0)\} \rightarrow \\ E \cup \{(\sum_i a_{1i} x_i + c_1 = 0), (\sum_i a_{2i} x_i + c_2 = 0)\} \cup \\ \{(\sum_i (k_1 a_{1i} + k_2 a_{2i}) x_i + (k_1 c_1 + k_2 c_2) = 0\}, \\ k_1, k_2 \in \mathbb{Z}. \end{array} \tag{5}$$

Decomposition of an equation

$$\begin{array}{r} E \cup \{\sum_i a_i x_i + c = 0\} \rightarrow E \cup \{\sum_i a_i x_i + c = 0\} \cup \\ \{(\sum_{i \neq k} a_i^q x_i + x_k - x_t + c^q = 0), (a_k x_t + \sum_{i \neq k} a_i^r x_i + c^r = 0)\} \\ \text{if } a_k = \text{argmin}_i \{|a_i| : a_i \neq 0\}, x_t \text{ is fresh,} \\ a_i = a_i^q a_k + a_i^r \text{ for all } i, \text{ and } c = c^q a_k + c^r. \end{array} \tag{6}$$

It is easy to see that Algorithm 1 implements a specific strategy of application of the above rules, namely:

- Step 3 corresponds to repeated applications of (4);
- Step 7 is an application of (5) multiple times; and
- Step 8 corresponds to an application of (6) followed by multiple applications of (5).

In order to generate explanations, we annotate each state of the above transition system with some additional information. In particular, let X be a set of variables containing all the variables in the initial equations and all the variables introduced by an application of (6), let L be a set of variables disjoint from X, and let λ be a mapping from variables in L to linear combinations of variables in X and of integer constants. Moreover, let σ be a partial mapping from variables in X to linear combinations of variables in X and of integer constants. An *annotated state* is a triple $\langle E', \lambda, \sigma \rangle$, where E' is a set of pairs $\langle e, \ell \rangle$ in which e is an equation and ℓ is a linear combination of variables from L. The initial state of the system is built as follows:

- $E' = \{\langle e_i, l_i \rangle : e_i \in E \text{ and } l_i \in L \text{ is fresh}\}$;

- For all $\langle e_i, l_i \rangle$ in E', set $\lambda(l_i) \mapsto e_i$;
- σ is initially empty.

We define inference rules for annotated states, corresponding to the rules (4)–(6):

Scaling of an equation

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell \rangle\}, \lambda, \sigma \rangle \rightarrow \\ \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell \rangle, \langle \sum_i \frac{a_i}{g} x_i + \frac{c}{g} = 0, \frac{1}{g}\ell \rangle\}, \lambda, \sigma \rangle \\ \text{if } g = GCD(a_i, \ldots, a_n, c) \text{ and } g > 1. \end{array} \tag{7}$$

Combination of two equations

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_{1i} x_i + c_1 = 0, \ell_1 \rangle, \langle \sum_i a_{2i} x_i + c_2 = 0, \ell_2 \rangle\}, \lambda, \sigma \rangle \rightarrow \\ \langle E' \cup \{\langle \sum_i a_{1i} x_i + c_1 = 0, \ell_1 \rangle, \langle \sum_i a_{2i} x_i + c_2 = 0, \ell_2 \rangle\} \cup \\ \{\langle \sum_i (k_1 a_{1i} + k_2 a_{2i}) x_i + (k_1 c_1 + k_2 c_2) = 0, k_1 \ell_1 + k_2 \ell_2 \rangle\}, \lambda, \sigma \rangle \\ k_1, k_2 \in \mathbb{Z}. \end{array} \tag{8}$$

Decomposition of an equation

$$\begin{array}{r} \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell \rangle\}, \lambda, \sigma \rangle \rightarrow \langle E' \cup \{\langle \sum_i a_i x_i + c = 0, \ell \rangle\} \cup \\ \{\langle \sum_{i \neq k} a_i^q x_i + x_k - x_t + c^q = 0, 0\ell \rangle, \\ \langle a_k x_t + \sum_{i \neq k} a_i^r x_i + c^r = 0, \ell \rangle\}, \lambda, \sigma' \rangle \\ \text{if } a_k = \text{argmin}_i\{|a_i| : a_i \neq 0\}, x_t \text{ is fresh,} \\ a_i = a_i^q a_k + a_i^r \text{ for all } i, c = c^q a_k + c^r \\ \text{and } \sigma'(x) = \begin{cases} \sum_{i \neq k} a_i^q x_i + x_k + c^q & \text{if } x = x_t \\ \sigma(x) & \text{otherwise.} \end{cases} \end{array} \tag{9}$$

The purpose of the variables in L and of the mapping λ is to give a name to each of the original equations. We observe in fact that at the beginning each equation is associated to a unique $l_i \in L$ through λ, and that none of the above rules modifies λ. Intuitively, each expression ℓ in a pair $\langle e, \ell \rangle$ of E' encodes the *linear combination of input equations* from which e was generated. When e is inconsistent, therefore, ℓ identifies exactly the subset of input equations responsible for the inconsistency. Analogously, when Algorithm 1 returns a solution S, each ℓ_i associated to an equation e_i in S identifies the subset of input equations which were used to generate the substitution encoded by e_i. This argument is formalized by the following theorem.

Theorem 2 *Let $\langle E', \lambda, \sigma \rangle$ be an annotated state. Let σ^* be the function that takes a linear combination and recursively replaces each fresh variable x_t introduced by (9) with $\sigma(x_t)$, until no more fresh variables are left. Finally, let λ^* be the function that takes a linear combination ℓ of variables from L and replaces each l with $\lambda(l)$. Then, for every element $\langle e, \ell \rangle$ of E', the following holds:*

$$\lambda^*(\ell) = \sigma^*(e). \tag{10}$$

Proof. First, observe that (10) holds for the initial state of the system, since each element of E' is in the form $\langle e_i, l_i\rangle$ such that $\lambda(l_i) = e_i$. We now show that any application of the rules (7)–(9) preserves (10).

In order to show that this is the case for (7) and (8), it is enough to observe that, for any linear combinations e_1 and e_2 and coefficients k_1 and k_2, $k_1\sigma^*(e_1) + k_2\sigma^*(e_2) = \sigma^*(k_1e_1 + k_2e_2)$, and similarly for λ^*.

As regards (9), let $e \overset{\text{def}}{=} \langle\sum_i a_ix_i + c = 0, \ell\rangle$ be the element that triggers the application of the rule, and suppose that (10) holds for it. Let

$$\langle e' \overset{\text{def}}{=} \sum_{i\neq k} a_i^q x_i + x_k - x_t + c^q = 0, 0\ell\rangle \text{ and} \tag{11}$$

$$\langle e'' \overset{\text{def}}{=} a_k x_t + \sum_{i\neq k} a_i^r x_i + c^r = 0, \ell\rangle \tag{12}$$

be the results of the decomposition, where x_t is fresh. Since (9) updates σ by setting $\sigma(x_t) \mapsto \sum_{i\neq k} a_i^q x_i + x_k + c^q$, by the definition of σ^* we have that:

(i)

$$\sigma^*(e') = \sigma^*(\sum_{i\neq k} a_i^q x_i + x_k + c^q - \sigma(x_t)) = \sigma^*(0) = 0,$$

which is clearly equal to $\lambda^*(0\ell) = 0$, and thus (10) holds for (11); and

(ii)

$$\sigma^*(e'') = \sigma^*(a_k\sigma(x_t) + \sum_{i\neq k} a_i^r x_i + c^r) =$$
$$\sigma^*(a_k(\sum_{i\neq k} a_i^q x_i + x_k + c^q) + \sum_{i\neq k} a_i^r x_i + c^r) = \sigma^*(e),$$

which is equal to $\lambda^*(\ell)$ by hypothesis. Therefore, (10) holds also for (12). □

Corollary 1 *Let $\langle e, \ell \overset{def}{=} \sum_i a_i l_i\rangle$ be an element of E', and let E_L be the set of equations $e_i = 0$ such that $\lambda(l_i) = e_i$ for each l_i in ℓ. If e is inconsistent, then E_L is inconsistent.*

Corollary 2 *Let S be a solution returned by Algorithm 1, let $S' \overset{def}{=} \{\langle e_i, \ell_i \overset{def}{=} \sum_j a_{ij} l_j\rangle \mid e_i \in S\}$ be its "annotated version", and let E_L be the set of equations $e_j = 0$ such that $\lambda(l_j) = e_j$ for each l_j in $\{\ell_i\}_i$. Let I be a set of inequalities, and let I' be the set of inequalities obtained from I after applying the substitutions in S and tightening the results. If I' is inconsistent, then $I \cup E_L$ is inconsistent.*

The two corollaries above give us a way of producing explanations with the Diophantine equation handler. Using Corollary 1, we can generate an explanation for an inconsistency detected directly by Algorithm 1 by taking the conjunction of all the input equations whose labels occur in the linear combination $\sum_i a_i l_i$ associated to the inconsistent equation e. Using Corollary 2, instead, we can identify, for each inequality generated by the equality elimination and tightening step, the set of equations used to generate it, by looking at the labels in the linear combinations $\sum_j a_{ij} l_j$ associated to each substitution e_i used. Thanks to this, we can generate an explanation for an inconsistency detected by the $\mathcal{LA}(\mathbb{Q})$-solver by first generating a $\mathcal{LA}(\mathbb{Q})$-explanation containing fresh inequalities generated by the elimination and tightening step, and then by replacing each of these fresh inequalities with the original inequality and the set of equations used to generate it.

4.2 Incrementality

It can be observed that our equality elimination procedure bears some similarities with algorithms for computing canonical forms of matrices over $\mathbb{Z}$, such as Hermite or Smith normal forms [22, 24]. In particular, our algorithm could be recast as a matrix manipulation procedure, which would then enable the use of very efficient matrix-based algorithms for its implementation, such as e.g. those given in [24].[2] An advantage of the formulation proposed here, however, is that it allows for efficient *incremental addition and removal* of equations. These features are very important in an SMT setting [23, 13], and it is not obvious how to integrate them efficiently into matrix-based algorithms. In contrast, the modifications needed for making Algorithm 1 fully incremental are very simple, and can be intuitively described as follows. Let E be a system of equations for which Algorithm 1 has already found a solution of S, and let e be a new equation not in E. Then, the consistency of $E \cup \{e\}$ can be checked by (i) applying the current substitutions in S to e, obtaining e'; and (ii) applying Algorithm 1 from Step 3 by setting F to be $\{e'\}$. If $E \cup \{e\}$ is inconsistent, then this is due to an equation that was generated from e' after some applications of Steps 7 or 8, possibly after having added some new substitutions to S. Then, the previous solution S for E can be recovered by simply dropping such substitutions. Similarly, if $E \cup \{e\}$ is consistent and S' is its solution, then S' is of the form $S \cup S''$, where S'' is the set of substitutions generated after the addition of e'. Therefore, also in this case after the removal of e the solution for E can be efficiently restored.

5. The Branch and Bound Module

When the equality elimination and tightening step does not lead to an inconsistency, the branch and bound module is activated. This module works by scanning the model produced by the $\mathcal{LA}(\mathbb{Q})$-solver in order to find integer variables that were assigned to a rational non-integer constant. If no such variable is found, then the $\mathcal{LA}(\mathbb{Q})$-model is also a $\mathcal{LA}(\mathbb{Z})$-model, and the solver returns sat. Otherwise, let x_k be an integer variable to which the $\mathcal{LA}(\mathbb{Q})$-solver has assigned a non-integer value q_k. Then, the branch and bound module (recursively) divides the problem in two subproblems obtained by adding respectively the constraints $(x_k - \lfloor q_k \rfloor \leq 0)$ and $(-x_k + \lceil q_k \rceil \leq 0)$ to the original formula, until either a $\mathcal{LA}(\mathbb{Z})$-model is found by the $\mathcal{LA}(\mathbb{Q})$-solver, or all the subproblems are proved unsatisfiable.

A popular approach for implementing this is to apply the splitting on-demand technique introduced in [2], by generating the $\mathcal{LA}(\mathbb{Z})$-lemma $(x_k - \lfloor q_k \rfloor \leq 0) \vee (-x_k + \lceil q_k \rceil \leq 0)$, and sending it back to the DPLL engine. The idea is that of exploiting the DPLL engine for the exploration of the branches introduced by branch and bound, rather than handling the case splits within the $\mathcal{LA}(\mathbb{Z})$-solver. As already observed in §2, this not only simplifies the

[2] We are grateful to an anonymous reviewer for this observation.

implementation, since there is no need of implementing support for disjunctive reasoning within the $\mathcal{LA}(\mathbb{Z})$-solver, but it also allows to exploit all the advanced search-space-reduction techniques implemented in modern DPLL engines.

However, using splitting on-demand has also some drawbacks. The first is that it does not easily allow to fully exploit the equality elimination and tightening step described in the previous section. Eliminating equalities introduces new integer variables and generates new inequalities which are *local* to the current $\mathcal{LA}(\mathbb{Z})$-solver call, and unknown to the DPLL engine. If we generate branch-and-bound lemmas from such internal state of the $\mathcal{LA}(\mathbb{Q})$-solver, there is a high risk that the generated lemmas will be only locally useful, since in our current implementation the tightened inequalities and the variables generated by the Diophantine equation handler are discarded upon backtracking. In fact, this is a more general problem of the splitting on-demand approach, even if no equality elimination is involved: the generation of all branch-and-bound lemmas is aimed at finding a $\mathcal{LA}(\mathbb{Z})$-model for the set of constraints *in the current DPLL branch*, and the lemmas might cease to be useful after backtracking.

Example 3 *Consider the following set of $\mathcal{LA}(\mathbb{Z})$-constraints:*

$$S \overset{def}{=} \begin{cases} x_1 - 2x_2 + 3x_3 = 0 \\ x_1 + 3x_3 + 1 \leq 0 \\ -2x_4 - x_1 + 1 \leq 0 \\ x_4 + x_3 \leq 0 \end{cases}$$

S is consistent in $\mathcal{LA}(\mathbb{Q})$. Suppose that the $\mathcal{LA}(\mathbb{Q})$-solver then returns the following model μ_S for it:

$$\mu_S \overset{def}{=} \begin{cases} x_1 = \frac{1}{5} \\ x_2 = -\frac{1}{2} \\ x_3 = -\frac{2}{5} \\ x_4 = \frac{2}{5} \end{cases}$$

After eliminating the first equation in S and tightening the result inequalities, the following system S' and $\mathcal{LA}(\mathbb{Q})$-model $\mu_{S'}$ are obtained: [3]

$$S' \overset{def}{=} \begin{cases} x_2 + 1 \leq 0 \\ -2x_4 - 2x_2 + 3x_3 + 1 \leq 0 \\ x_4 + x_3 \leq 0 \end{cases} \qquad \mu_{S'} \overset{def}{=} \begin{cases} x_2 = -1 \\ x_3 = -\frac{3}{5} \\ x_4 = \frac{3}{5} \end{cases}$$

Suppose that the branch and bound module selects the variable x_4, and that the first branch to be explored is that corresponding to the addition of the constraint $(x_4 - \lfloor\frac{3}{5}\rfloor \leq 0)$. If the starting point of branch and bound is S', then a $\mathcal{LA}(\mathbb{Z})$-model can be found immediately by the $\mathcal{LA}(\mathbb{Q})$-solver (after adding $(x_4 - \lfloor\frac{3}{5}\rfloor \leq 0)$). If branch and bound is implemented via splitting on-demand, however, the branch and bound exploration is started from the original set of constraints S: in this case, adding the constraint $(x_4 - \lfloor\frac{3}{5}\rfloor \leq 0)$ is not enough to find a

[3] $\mu_{S'}$ is actually what MathSAT5 produces when starting from the model μ_S, following the Simplex algorithm as given in [13].

$\mathcal{LA}(\mathbb{Z})$-model (with the algorithm of [13]), and the following $\mathcal{LA}(\mathbb{Q})$-model μ'' is generated:

$$\mu'' \stackrel{\text{def}}{=} \begin{cases} x_1 = 1 \\ x_2 = -\frac{1}{2} \\ x_3 = -\frac{2}{3} \\ x_4 = 0 \end{cases}$$

In order to find a $\mathcal{LA}(\mathbb{Z})$-model, therefore, more search is needed. ◇

A second issue is that of *non-chronological* backtracking. While this feature is crucial for the performance of modern DPLL-based SAT solvers as it in general allows to significantly prune the search space, as already observed in [23] in an SMT context it might sometimes hurt performance, in particular for satisfiable problems. Suppose that the conjunction of constraints μ corresponding to the current branch is $\mathcal{LA}(\mathbb{Z})$-satisfiable, but the current $\mathcal{LA}(\mathbb{Q})$-model value q_k for the integer variable x_k is not an integer, and suppose that adding $(-x_k+\lceil q_k\rceil \leq 0)$ allows the $\mathcal{LA}(\mathbb{Q})$-solver to find a $\mathcal{LA}(\mathbb{Z})$-model, but adding $(x_k-\lfloor q_k\rfloor \leq 0)$ results in a $\mathcal{LA}(\mathbb{Q})$-inconsistency. Then, if DPLL branches on $(x_k - \lfloor q_k\rfloor \leq 0)$ first, non-chronological backtracking might undo the assignments of a (potentially large) subset of the literals in μ, which would then have to be re-assigned and re-sent to the $\mathcal{LA}(\mathbb{Q})$-solver. In the worst case, after backjumping DPLL might flip the truth value of some of the literals in μ, possibly resulting in more conflicts before finding the $\mathcal{LA}(\mathbb{Z})$-consistent truth-assignment μ again.

Finally, the use of splitting on-demand makes it more complex to use *dedicated heuristics* for exploring the branch-and-bound search tree. It is well-known in the Integer Programming community that a careful selection of the variables on which to perform case splits, based on information provided by the $\mathcal{LA}(\mathbb{Q})$-solver, can have a significant impact on performance (see e.g. [1]). With splitting on-demand, such heuristics would need to be integrated with those commonly used in DPLL, which might not be straightforward.

In principle, these drawbacks could be addressed by carefully modifying the heuristics used by the DPLL engine for variable selection, backjumping, and management of learnt clauses. However, integrating dedicated $\mathcal{LA}(\mathbb{Z})$-specific heuristics in the "general" DPLL engine might not only be difficult to implement, but it also might introduce inefficiencies and/or hard-to-understand interactions with the generic heuristics commonly used in SAT. Therefore, we have adopted a different solution, which is based on a mechanism to handle the branch-and-bound search *within the $\mathcal{LA}(\mathbb{Z})$-solver itself*, without the intervention of the DPLL engine. This "internal" branch and bound submodule is invoked only for a bounded (and small) number of case splits, [4.] and does not perform any non-chronological backtracking. In our experiments, we have seen that for many satisfiable problems only a few branch-and-bound case splits are enough to find a $\mathcal{LA}(\mathbb{Z})$-model, especially if the "right" variables are selected. Performing such case splits internally makes it much easier to implement different heuristics for variable selection. In our current implementation, we use a history-based greedy strategy, which selects the variable that resulted in the minimum number of violations of integrality constraints in the previous branches, inspired

[4.] In our actual implementation, we use a bound that is proportional to the number of variables in the input problem.

by the "pseudocost branching" rule described in [1]. More specifically, let n_k^l and n_k^r be respectively the number of left and right branches on the variable x_k in the branch and bound search, [5] and let $(g_k^l)_i$ and $(g_k^r)_i$ be respectively the number of integer variables with non-integer values after having performed the i-th left (resp. right) branch on x_k. Then, the *score* of x_k is defined as the minimum between $(\sum_{i=1}^{n_k^l}(g_k^l)_i)/n_k^l$ and $(\sum_{i=1}^{n_k^r}(g_k^r)_i)/n_k^r$, and our heuristic always selects the variable with the smallest score.

Finally, another advantage of using an internal branch and bound search is that it also allows to perform some simplifications of the current set of constraints (before starting the internal branching) which can significantly help in finding a $\mathcal{LA}(\mathbb{Z})$-model. In particular, we currently try to detect and remove redundant constraints from the $\mathcal{LA}(\mathbb{Q})$-solver before starting the branch-and-bound search.

If the "internal" branch and bound finds a conflict, an explanation can be recursively generated by combining, for each node of the branch-and-bound tree, the $\mathcal{LA}(\mathbb{Q})$-explanations for the conflicts on the two sub-branches.[6] If the limit on the number of case splits is reached, then the splitting on-demand approach is used, generating a branch-and-bound lemma and sending it to the DPLL engine. This allows us to keep the good features of the splitting on-demand approach for problems that cannot be easily solved with a few branch-and-bound case splits.

5.1 Adding Cuts from Proofs

A severe drawback of branch and bound is that it might fail to terminate (continuing to generate new branch-and-bound lemmas) if the input problem contains some unbounded variable. This is not "simply" a theoretical possibility, but it also happens quite often in practice, even for very small and simple problems.

Example 4 *Consider the following set of $\mathcal{LA}(\mathbb{Z})$-constraints:*

$$S \overset{def}{=} \begin{cases} 5x_1 - 5x_2 - x_3 - 3 \leq 0 \\ -5x_1 + 5x_2 + x_3 + 2 \leq 0 \\ x_3 \leq 0 \\ -x_3 \leq 0 \end{cases}$$

It is easy to see that S is $\mathcal{LA}(\mathbb{Z})$-inconsistent: by projecting out x_3, we obtain

$$S' \overset{def}{=} \begin{cases} 5x_1 - 5x_2 - 3 \leq 0 & \textit{which can be tightened to} \quad x_1 - x_2 \leq 0 \\ -5x_1 + 5x_2 + 2 \leq 0 & \textit{which can be tightened to} \quad -x_1 + x_2 + 1 \leq 0 \end{cases}$$

which is trivially $\mathcal{LA}(\mathbb{Q})$-inconsistent. However, the application of branch and bound to S results in an infinite branch-and-bound tree, because for every integer model value for x_1 (resp. x_2) there exists a non-integer value for x_2 (resp. x_1) that $\mathcal{LA}(\mathbb{Q})$-satisfies the first two constraints of S (the $\mathcal{LA}(\mathbb{Q})$-model for x_3 will always be zero due to the last two constraints of S). ◇

[5] Here, we call left branch the branch on $(x_k - \lfloor q_k \rfloor \leq 0)$, and right branch that on $(-x_k + \lceil q_k \rceil \leq 0)$.

[6] More specifically, if $\neg\eta_l \wedge (x_k - \lfloor q_k \rfloor \leq 0)$ is the $\mathcal{LA}(\mathbb{Q})$-explanation of the left branch and $\neg\eta_r \wedge (-x_k + \lceil q_k \rceil \leq 0)$ is the $\mathcal{LA}(\mathbb{Q})$-explanation of the right branch, then $\neg\eta_l \wedge \neg\eta_r$ is the $\mathcal{LA}(\mathbb{Z})$-explanation of the current node.

A common approach for overcoming this limitation is that of complementing branch and bound with the generation of *cutting planes* [1, 22], which are inequalities that exclude some $\mathcal{LA}(\mathbb{Q})$-models of the current set of constraints without losing any of its $\mathcal{LA}(\mathbb{Z})$-models. In particular, Gomory's cutting planes (Gomory cuts) are very often used in practice [14, 1].[7]

In our $\mathcal{LA}(\mathbb{Z})$-solver, instead, we follow a different approach, which has been recently proposed in [12] and shown to outperform SMT solvers based on branch and bound with Gomory cuts. The idea of the algorithm is that of extending the branch and bound approach to split cases not only on individual variables, but also on more general linear combinations of variables, thus generating lemmas like $(\sum_k a_k x_k - \lfloor q_k \rfloor \leq 0) \vee (-\sum_k a_k x_k + \lceil q_k \rceil \leq 0)$.

The core of the cuts from proofs algorithm is the identification of the *defining constraints* of the current solution of the rational relaxation of the input set of $\mathcal{LA}(\mathbb{Z})$-constraints. A defining constraint is an input constraint $\sum_i c_i x_i + c \bowtie 0$ (where $\bowtie \in \{\leq, =\}$) such that $\sum_i c_i x_i + c$ evaluates to zero under the current solution for the rational relaxation of the problem. After having identified the defining constraints D, the cuts from proofs algorithm checks the satisfiability of the system of Diophantine equations $D_E \stackrel{\text{def}}{=} \{\sum_i c_i x_i + c = 0 \mid (\sum_i c_i x_i + c \bowtie 0) \in D\}$. If D_E is unsatisfiable, then it is possible to generate a proof of unsatisfiability for it, expressed as a $\mathcal{LA}(\mathbb{Z})$-inconsistent linear combination of a subset of the constraints in D_E, whose result is an equation $\sum_i c'_i x_i + c' = 0$ such that the GCD g of the c'_i's does not divide c'. From this equation, the following extended branch and bound lemma is generated:

$$(\sum_i \frac{c'_i}{g} x_i - \left\lfloor \frac{-c'}{g} \right\rfloor \leq 0) \vee (-\sum_i \frac{c'_i}{g} x_i + \left\lceil \frac{-c'}{g} \right\rceil \leq 0).$$

In the original algorithm of [12], proofs of unsatisfiability are generated by computing Hermite Normal Forms [22]. However, this is not necessary: in our solver, in fact, we can reuse the module for handling Diophantine equations, thanks to its proof-production capability. This makes the implementation very simple.

Example 5 *Consider again the set S of $\mathcal{LA}(\mathbb{Z})$-constraints of Example 4, and suppose that the $\mathcal{LA}(\mathbb{Q})$-solver returns the following $\mathcal{LA}(\mathbb{Q})$-model for S:*

$$\mu_S \stackrel{\text{def}}{=} \begin{cases} x_1 = 0 \\ x_2 = -\frac{2}{5} \\ x_3 = 0 \end{cases}$$

The set of defining constraints D for $\langle S, \mu_S \rangle$ is then:

$$D \stackrel{\text{def}}{=} \begin{cases} -5x_1 + 5x_2 + x_3 + 2 \leq 0 \\ x_3 \leq 0 \\ -x_3 \leq 0, \end{cases}$$

resulting in the following inconsistent system of Diophantine equations D_E:

$$D_E \stackrel{\text{def}}{=} \begin{cases} -5x_1 + 5x_2 + x_3 + 2 = 0 \\ x_3 = 0 \end{cases}$$

[7] Gomory cuts guarantee termination provided that the strategies for pivoting in the Simplex and for generating the cuts satisfy certain conditions, which are given e.g. in [22].

The Diophantine equations handler generates $-5x_1+5x_2+2=0$ *as proof of unsatisfiability of* D_E*, resulting in the following branch-and-bound lemma:*

$$(-x_1+x_2+1 \leq 0) \vee (x_1-x_2 \leq 0). \tag{13}$$

After adding (13) *to DPLL, the* $\mathcal{LA}(\mathbb{Q})$*-solver detects the* $\mathcal{LA}(\mathbb{Q})$*-inconsistency of both* $S \cup (-x_1+x_2+1 \leq 0)$ *and* $S \cup (x_1-x_2 \leq 0)$*.* ◇

For more information on the cuts from proofs algorithm, we refer the reader to [12]. Here we only mention that, as already observed in [12], we found that in practice it is a good idea to interleave the generation of "extended" branches with that of "regular" branches on individual variables. Currently, we use a simple heuristic in which an extended branch is tried only after two regular branches. The investigation of alternative strategies is part of future work.

5.2 Termination of the Branch and Bound Module

In order to prove the termination for the cuts from proofs algorithm, the authors of [12] assume that all the input variables are bounded. From the theoretical point of view, this is not a limitation, since as shown e.g. in [22] it is always possible to statically determine bounds for all unbounded variables of an arbitrary set of $\mathcal{LA}(\mathbb{Z})$-constraints. In fact, by adding such bounds it is also possible to guarantee the termination of the "regular" branch and bound technique. However, not only are such bounds in general so large to have no practical value for ensuring termination within reasonable amounts of time and space [14], but they might also be detrimental for performance, as they would force the $\mathcal{LA}(\mathbb{Z})$-solver to manipulate very large numbers requiring arbitrary-precision arithmetic, which can otherwise often be avoided in practice [17]. The advantage of using cuts from proofs over regular branch and bound is that with the former the need of such theoretical bounds is *much less likely* in practice.[8.]

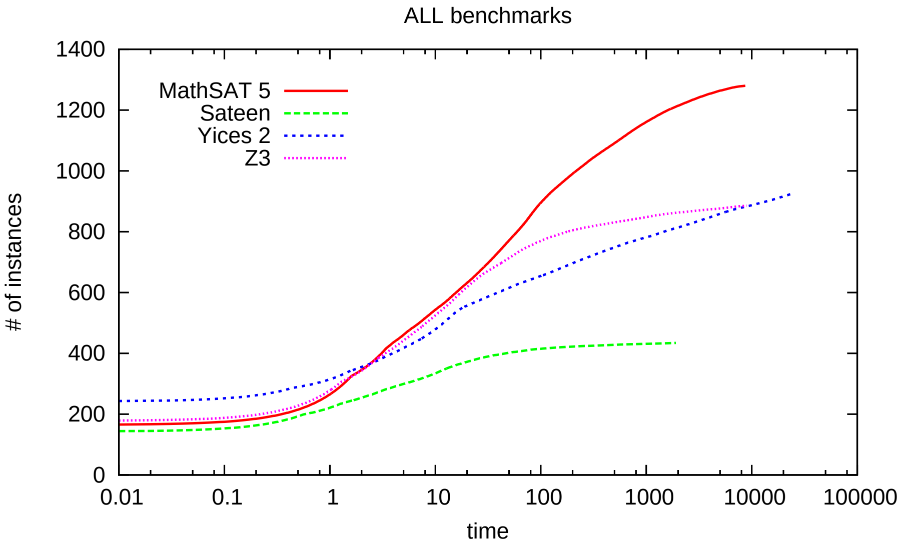
For these reasons, in our current implementation we do not precompute bounds for all the unbounded variables in the input problem. Therefore, in principle our solver might fail to terminate. In order to overcome this limitation, we plan to investigate the possibility of combining the cuts from proofs algorithm with other techniques (like Gomory cuts or the recently-proposed algorithm of [19]) which ensure termination without requiring precomputed bounds for all the variables.

6. Experiments

We have implemented the $\mathcal{LA}(\mathbb{Z})$-solver presented here within our new SMT solver MathSAT5. In the latest SMT solvers competition SMT-COMP 2010,[9.] MathSAT5 ranked first in both divisions involving $\mathcal{LA}(\mathbb{Z})$, QF_LIA (quantifier-free $\mathcal{LA}(\mathbb{Z})$) and QF_UFLIA (quantifier free $\mathcal{LA}(\mathbb{Z})$ with uninterpreted functions), solving significantly more benchmarks than the other competitors. In this section, we perform a more detailed experimental evaluation

[8.] In fact, we have not been able to conceive a simple example for which the cuts from proofs algorithm needs precomputed bounds.

[9.] http://smtcomp.org/2010/



Solver	# solved	total time
MathSAT5	1281 / 1312	8677.80
Yices2	928 / 1312	25421.15
Z3	887 / 1312	8983.43
Sateen	435 / 1312	1908.38

Figure 3. Comparison among different SMT solvers, all benchmarks.

of its performance. We have run the experiments on a machine with a 2.6 GHz Intel Xeon processor, 16 GB of RAM and 6 MB of cache, running Debian GNU/Linux 5.0. We have used a time limit of 600 seconds and a memory limit of 2 GB.

6.1 Description of the benchmark instances

We concentrate our evaluation on a subset of the problems in the QF_LIA category of the library of SMT problems SMT-LIB [10]. Out of the more than 5200 problems in this category, we have selected the subset of instances for which either the $\mathcal{LA}(\mathbb{Q})$-solver is not enough for deciding satisfiability, or which cannot be greatly simplified by applying some ad-hoc preprocessing techniques (like e.g. [18]), in order to test the effectiveness of $\mathcal{LA}(\mathbb{Z})$-specific algorithms. In particular, we use the following families of benchmarks:

CAV09, which are the randomly-generated conjunctions of $\mathcal{LA}(\mathbb{Z})$-inequalities which were used in [12]. Most of the instances are satisfiable.

[10] http://smtlib.org

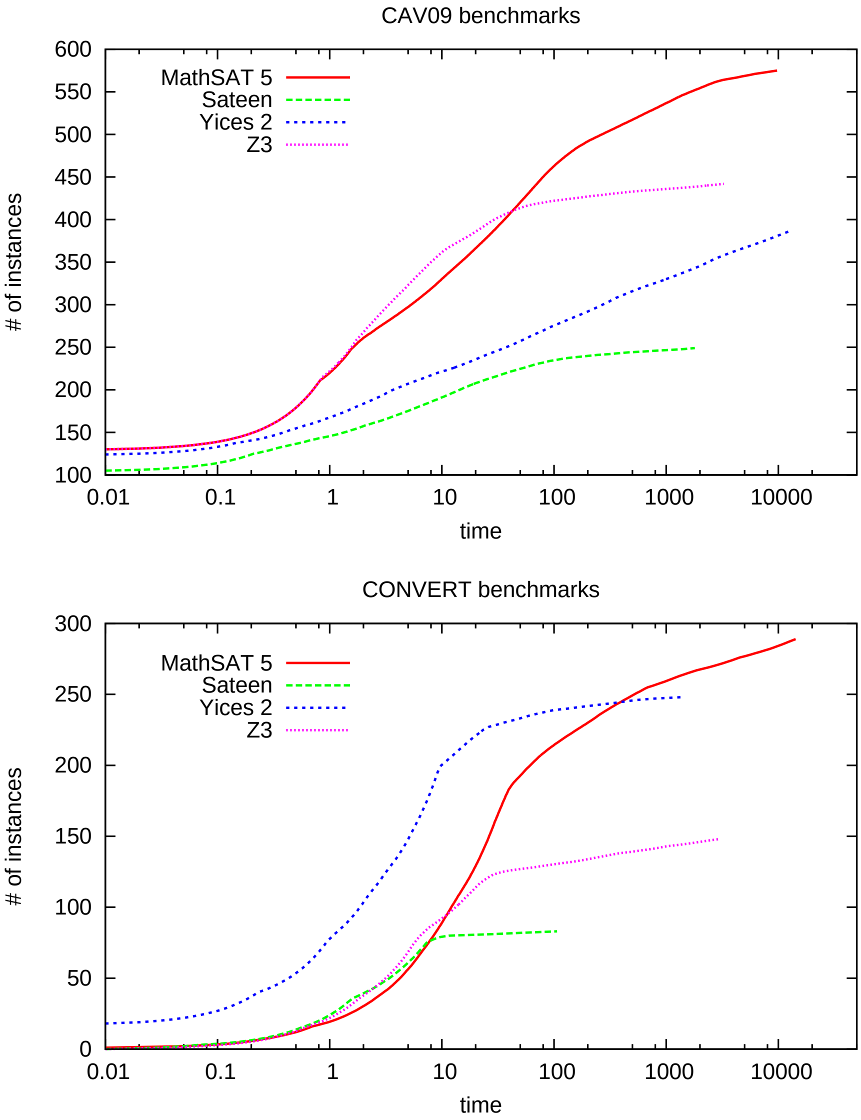


Figure 4. Comparison among different SMT solvers (CAV09 and CONVERT benchmarks).

CONVERT, which have been obtained by encoding in SMT($\mathcal{LA}(\mathbb{Z})$) some bit-vector formulas from the QF_BV category of the SMT-LIB, using the encoding of [6].[11] Most of the instances are satisfiable.

CUT_LEMMAS, which are crafted instances encoding the $\mathcal{LA}(\mathbb{Z})$-validity of some cutting planes. All the instances are unsatisfiable.

RINGS_PREPROCESSED, which are crafted instances checking distributivity and associativity of addition and multiplication in modular arithmetic. They are prepro-

[11] The non-easily-linearizable operators have been replaced with fresh variables.

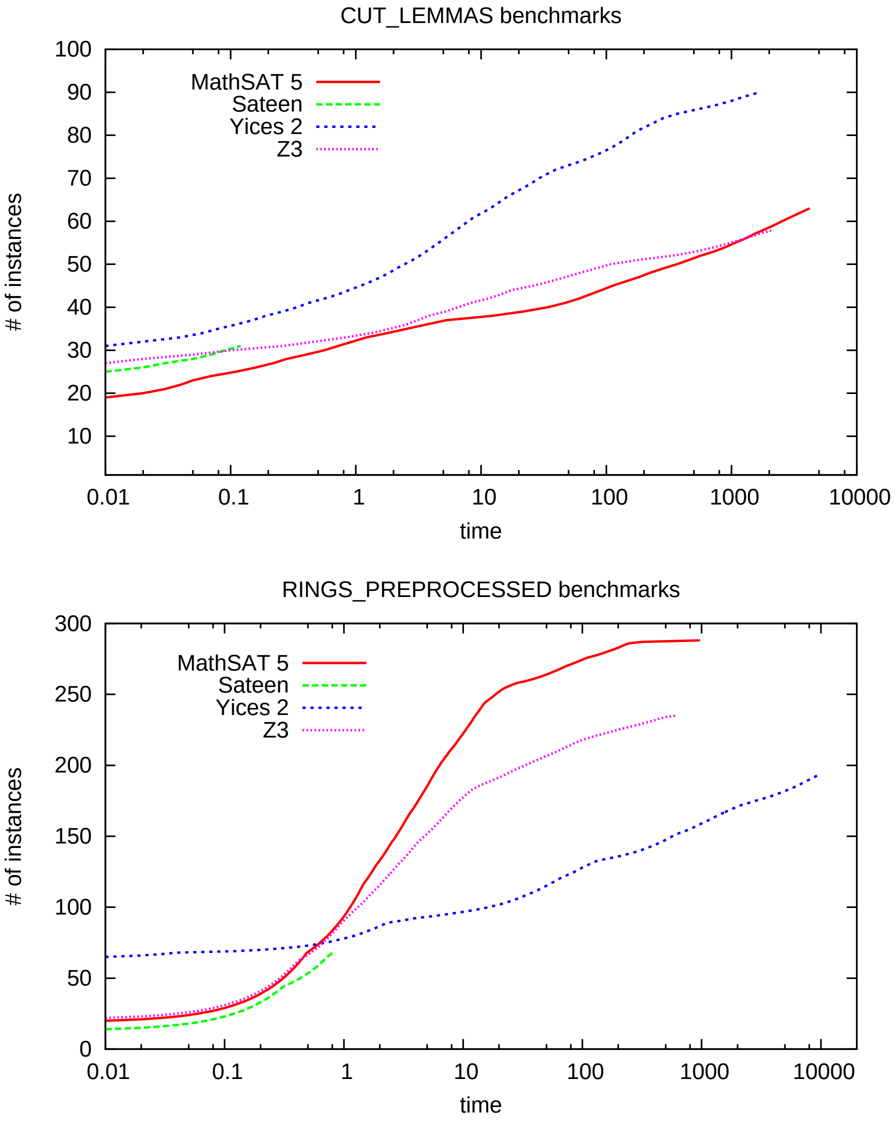
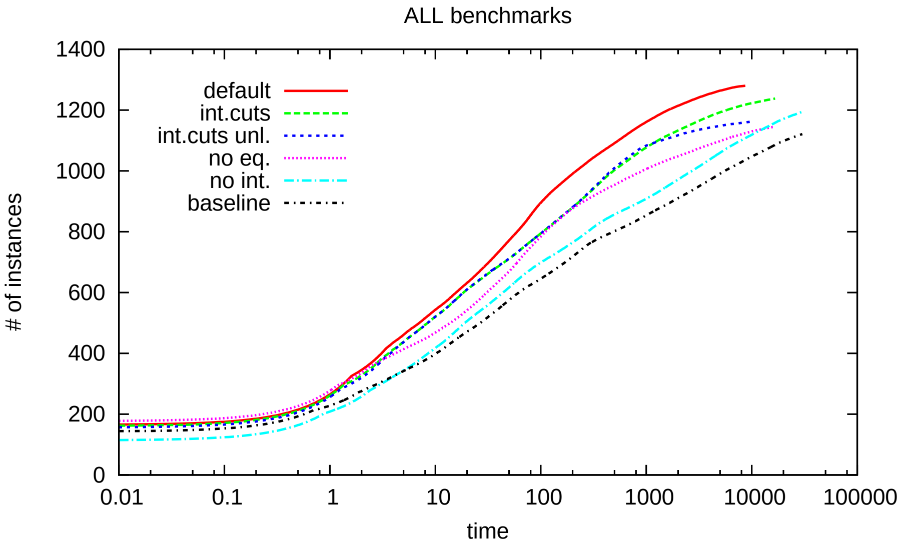


Figure 5. Comparison among different SMT solvers (CUT_LEMMAS and RINGS_PREPROCESSED benchmarks).

cessed versions of the `rings` instances in the SMT-LIB, in which all the term-level if-then-else constructs have been eliminated in a simple way, [12] in order to prevent the application of the preprocessing technique of [18]. All the instances are unsatisfiable.

In total, our benchmark set consists of 1312 instances.

[12] Each $ite(c, t, e)$ has been replaced by a fresh variable v, and the constraint $(c \to v = t) \wedge (\neg c \to v = e)$ has been added to the formula.



Configuration	# solved	total time
default	1281 / 1312	8677.80
int.cuts	1239 / 1312	16670.56
no int.	1194 / 1312	29559.84
int.cuts unl.	1163 / 1312	9949.57
no eq.	1146 / 1312	16256.53
baseline	1122 / 1312	30110.24

Configurations:

‘default’: all heuristics enabled
‘no int.’: disable internal b.&b.
‘no eq.’: disable equality elim.
‘int.cuts’: use cuts from proofs in the internal b.&b.
‘int.cuts unl.’: internal b.&b. w/o timeout, with cuts
‘baseline’: disable equality elim. and internal b.&b.

Figure 6. Comparison among different $\mathcal{LA}(\mathbb{Z})$-heuristics in MathSAT5 (all benchmarks).

6.2 Evaluation

In the first part of our evaluation, we compare MathSAT5 with state-of-the-art SMT solvers for $\mathcal{LA}(\mathbb{Z})$, namely Sateen [18] (winner of the 2009 SMT-COMP competition on QF_LIA), Z3 [11] (winner of 2008 [13]) and Yices2 (the new version of the popular Yices solver [13]). [14] The results are collected in Figures 3–5. The plots show the accumulated time (on the X axis) for solving a given number of instances (on the Y axis) within the timeout, both for each individual benchmark family as well as for all the benchmarks. They show that, with the exception of the CUT_LEMMAS family on which Yices2 has very good performances, MathSAT5 outperforms the other solvers: overall, MathSAT5 can solve about 38% more problems than the closest competitor Yices2, with a significantly shorter

[13] We have however used version 2.4 of Z3, which is newer than the one of SMT-COMP’08.

[14] We would have liked to compare also with the tool of [12], but it was not possible to obtain it.

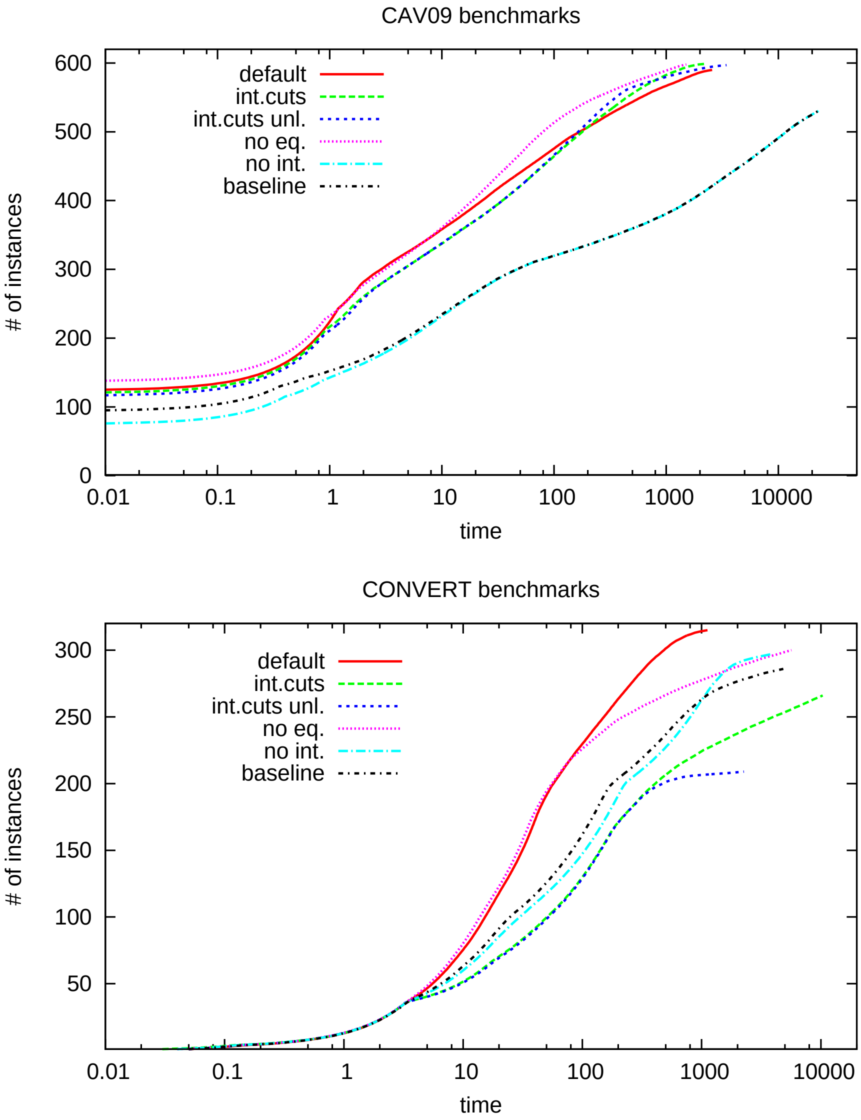


Figure 7. Comparison among different $\mathcal{LA}(\mathbb{Z})$-heuristics in MathSAT5 (CAV09 and CONVERT benchmarks).

total execution time. We remark that MathSAT5 implements the same $\mathcal{LA}(\mathbb{Q})$-procedure of [13] as Z3 and Yices2, and its performance on SMT($\mathcal{LA}(\mathbb{Q})$) is comparable to that of these two solvers.

In the second part of our experiments, we compare different configurations of MathSAT5, in order to evaluate the effectiveness of our techniques and heuristics. The results are shown in Figures 6–8. We ran MathSAT5 with equality elimination and tightening disabled ('no eq.'), with the internal branch and bound disabled ('no int.'), with both disabled ('baseline'), using the cuts from proofs technique of [12] also in the internal branch

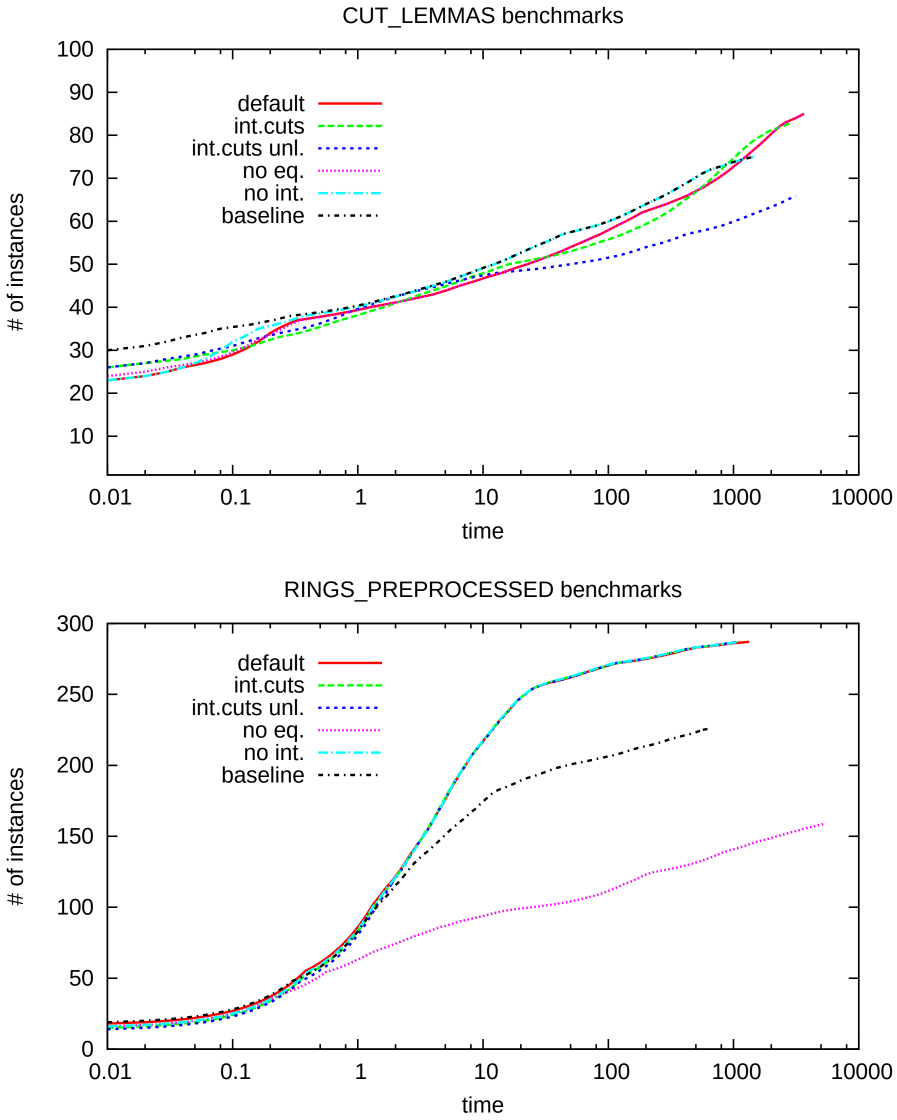


Figure 8. Comparison among different $\mathcal{LA}(\mathbb{Z})$-heuristics in MathSAT5 (CUT_LEMMAS and RINGS_PREPROCESSED benchmarks).

and bound, and not only in splitting on-demand ('int.cuts'), and with an unlimited internal branch and bound ('int. cuts unl.', thus effectively disabling splitting on-demand). The results show that all the techniques and heuristics described in this paper contribute to the performance of the solver. The default configuration is not the best one only for the CAV09 family, for which applying the cuts from proofs technique of [12] more eagerly allows to solve 9 more instances (out of 600). However, this worsens performance in general, especially on the instances of the CONVERT family.

7. Conclusions

We have presented a new theory solver for Linear Integer Arithmetic in a lazy SMT context, whose distinguishing feature is an extensive use of layering and heuristics for combining different techniques. Our experimental evaluation demonstrates the potential of the approach, showing significant improvements on a variety of benchmarks wrt. approaches used in current state-of-the-art SMT solvers.

References

[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In M. Hermann and A. Voronkov, editors, *Proceedings of LPAR'06*, **4246** of *LNCS*, pages 512–526. Springer, 2006.

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, **4590** of *LNCS*, pages 298–302. Springer, 2007.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, **185** of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[5] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez Carbonell, and A. Rubio. The Barcelogic SMT Solver. In A. Gupta and S. Malik, editors, *Proceedings of CAV'08*, **5123** of *LNCS*, pages 294–298. Springer, 2008.

[6] 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. *Electr. Notes Theor. Comput. Sci.*, **144**(2):3–14, 2006.

[7] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT5: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, **35**(1-3):265–293, 2005.

[8] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In W. Damm and H. Hermanns, editors, *Proceedings of CAV'07*, **4590** of *LNCS*. Springer, 2007.

[9] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of TACAS*, **6015** of *LNCS*, pages 150–153. Springer, 2010.

[10] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, **5**(7):394–397, 1962.

[11] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of TACAS'08*, **4963** of *LNCS*, pages 337–340. Springer, 2008.

[12] I. Dillig, T. Dillig, and A. Aiken. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In A. Bouajjani and O. Maler, editors, *Proceedings of CAV'09*, **5643** of *LNCS*. Springer, 2009.

[13] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Proceedings of CAV'06*, **4144** of *LNCS*, pages 81–94. Springer, 2006.

[14] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report CSL-06-01, SRI, 2006.

[15] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 2^{nd} edition, 2001.

[16] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez Carbonell. SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In Hans Kleine Buning and Xishun Zhao, editors, *Proceedings of SAT'08*, **4996** of *LNCS*, pages 77–90. Springer, 2008.

[17] A. Griggio. *An Effective SMT Engine for Formal Verification.* PhD thesis, DISI, University of Trento, 2009.

[18] H. Kim, F. Somenzi, and H. Jin. Efficient Term-ITE Conversion for Satisfiability Modulo Theories. In O. Kullmann, editor, *Proceedings of SAT'09*, **5584** of *LNCS*, pages 195–208. Springer, 2009.

[19] D. Kroening, J. Leroux, and P. Rümmer. Interpolating Quantifier-Free Presburger Arithmetic. In Christian G. Fermüller and Andrei Voronkov, editors, *Proceedings of LPAR-17*, **6397** of *LNCS*, pages 489–503. Springer, 2010.

[20] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, **53**(6):937–977, 2006.

[21] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of SC*, pages 4–13, 1991.

[22] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[23] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, **3**(3-4):141–224, 2007.

[24] A. Storjohann. *Algorithms for Matrix Canonical Forms.* PhD thesis, Swiss Federal Institute of Technology – ETH, 2000.

[25] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In *Proceedings of CAV'02*, number 2404 in LNCS, pages 17–36. Springer, 2002.