A Practical Approach to SMT($\mathcal{LA}(\mathbb{Z})$)

Alberto Griggio

FBK-IRST, Trento, Italy. griggio@fbk.eu

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.

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 [6, 8]. 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 [5], this indicates that the theory solvers for $\mathcal{LA}(\mathbb{Z})$ currently used (e.g. [7, 11]), although theoretically complete, are not very robust in practice.

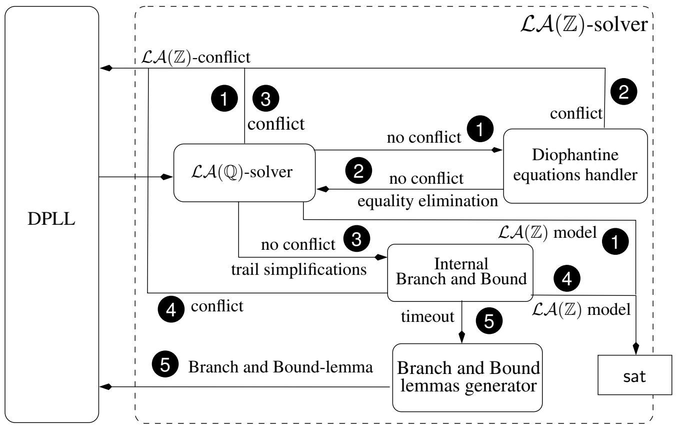
In this paper, we present a new theory solver for $\mathcal{LA}(\mathbb{Z})$ which explicitly focuses on achieving good performance on practical examples. They key feature of our solver is an extensive use of *layering* and *heuristics* for combining different known techniques, in order to exploit the 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 solver in a lazy SMT system based on the DPLL(T) architecture. Finally, we demonstrate the viability of our approach by evaluating our implementation 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 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 §3 and §4. In §5 we describe the experimental evaluation, and in §6 we conclude. Due to the scope of the workshop and to lack of space, we shall assume familiarity with the DPLL(T) architecture and the concepts and terminology commonly used in lazy SMT, which can be found e.g. in [13].

2 General Architecture

Figure 1 shows an outline of the general architecture of the $\mathcal{LA}(\mathbb{Z})$-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.



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

Figure 1: Architecture of the $\mathcal{LA}(\mathbb{Z})$-solver.

First, the real relaxation of the problem is checked, using a Simplex-based $\mathcal{LA}(\mathbb{Q})$-solver similar to that described in [6]. 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 [11]: 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 parametric 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 [12]. 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 $(t_1 \neq t_2) \rightarrow (t_1 < t_2) \vee (t_1 > t_2)$ for each disequality $(t_1 \neq t_2)$ sent to the $\mathcal{LA}(\mathbb{Z})$-solver.

3 Equality Elimination

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 [11].

Given a system $E \overset{\text{def}}{=} \{\sum_{i=1}^{n} a_{ji}x_i + c_j = 0\}_{j=1}^{m}$ 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 [11], and can be easily made incremental.

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 possible to *tighten* some of the new inequalities generated. Given an inequality such that the GCD g of the a_i's does not divide the constant c, a tightening step [11] consists in rewriting it into $\sum_i \frac{a_i}{g} x_i \leq \lfloor \frac{c}{g} \rfloor$. Tightening is important because it might allow the $\mathcal{LA}(\mathbb{Q})$-solver to detect more conflicts.

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 [6], our $\mathcal{LA}(\mathbb{Q})$-solver does not assume that only *elementary bounds* of the form $(x \leq c)$ are asserted and retracted during search, but rather it supports the addition and deletion of arbitrary constraints. [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

[1] The distinction between tableau equations and elementary bounds introduced in [6] is still used, but such transformation is performed internally in the $\mathcal{LA}(\mathbb{Q})$-solver rather than at preprocessing time [9].

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 $F = E, S = \emptyset$.
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., [11]); return unsat.
5. Otherwise, let $e_h \overset{\text{def}}{=} \sum_i a_{hi}x_i + c_h = 0$ be an equation, and let $a_{hk}$ be the non-zero
   coefficient with the smallest absolute value in $e_h$.
6. If $|a_{hk}| = 1$, then $e_h$ can be rewritten as
       $$-x_k + \sum_{i \neq k} -\text{sign}(a_{hk})a_{hi}x_i - \text{sign}(a_{hk})c_h = 0,$$
   where $\text{sign}(a_{hk}) \overset{\text{def}}{=} \frac{a_{hk}}{|a_{hk}|}$. Then, remove $e_h$ from $F$, add it to $S$, and replace $x_h$
   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 $F$.
7. If $|a_{hk}| > 1$, then rewrite $e_h$ as
       $$a_{hk}x_k + \sum_{i \neq k}(a_{hk}a^q_{hi} + a^r_{hi})x_i + (a_{hk}c^q_h + c^r_h) \;= 0 \;\equiv$$
       $$a_{hk} \cdot (x_k + \sum_{i \neq k} a^q_{hi}x_i + c^q_h) + (\sum_{i \neq k} a^r_{hi}x_i + c^r_h) = 0.$$
   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 $F$.
8. Go to Step 2.
```

the tightening of the inequalities, in order to express the conflict clause in terms of constraints that are known to the DPLL engine.

In order to generate explanations, we introduce a special *label variable* l_j for each input equation e_j, which we use to keep track of the operations performed in the various steps of Algorithm 1. Each input equation $e_j \overset{\text{def}}{=} (\sum_i a_{ij}x_i + c_j = 0)$ becomes $(l_j + \sum_i a_{ij}x_i + c_j = 0)$. Each equation generated during a run of Algorithm 1 (in Steps 3, 6 and 7) is then of the form $(\sum_k b_{kj}l_k + \sum_i a_{ij}x_i + c_j =$

$0)$. When inconsistency is detected (Step 4), an explanation can be generated by simply taking each input equation e_j whose lable variable l_j occurs with non-zero coefficient in the conflicting equation. In fact, the expression $\sum_k b_{kj} l_k$ constitutes a *proof of unsatisfiability* of the system of equations, which tells us exactly *how* to combine the input equations in order to obtain an inconsistency. Explanations for the equations e_j in the solution S returned when no inconsistency is detected are generated in the same way.

For lack of space, we can not provide the details of how to extend the Steps of Algorithm 1 in order to take into account label variables. For this, we refer to [9], where the procedure is formally described and proved correct.

4 Branch and Bound

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 $(x_k \leq \lfloor q_k \rfloor)$ and $(x_k \geq \lceil q_k \rceil)$ 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 $(x_k \leq \lfloor q_k \rfloor) \vee (x_k \geq \lceil q_k \rceil)$, 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. This not only simplifies the implementation, since there is no need of implementing support for disjunctive reasoning within the $\mathcal{LA}(\mathbb{Z})$-solver, 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.

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.

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 [13] 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 $(x_k \geq \lceil q_k \rceil)$ allows the $\mathcal{LA}(\mathbb{Q})$-solver to find a $\mathcal{LA}(\mathbb{Z})$-model, but adding $(x_k \leq \lfloor q_k \rfloor)$ results in a $\mathcal{LA}(\mathbb{Q})$-inconsistency. Then, if DPLL branches on $(x_k \leq \lfloor q_k \rfloor)$ 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 order to address the above issues, we have implemented 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, 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 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 the branch and bound search, [2] and let $(g_k^l)_i$ and $(g_k^r)_i$ be respectively the number of integer variables with non-integer value 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

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

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 easily generated by resolution. 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.

Adding cuts from proofs. An important point to highlight is that branch and bound might fail to terminate (continuing to generate new branch-and-bound lemmas) if the input problem contains some unbounded variable. Although theoretically it is possible to statically determine bounds for all unbounded variables and thus to make branch and bound complete [12], such theoretical bounds would be so large to have no practical value [7].

A common approach for overcoming this limitation is that of complementing branch and bound with the generation of *cutting planes* [1, 12], 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 [7, 1].

In our $\mathcal{LA}(\mathbb{Z})$-solver, instead, we follow a different approach, which has been recently proposed in [5] 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 $(\sum_k a_k x_k \leq \lfloor q_k \rfloor) \vee (\sum_k a_k x_k \geq \lceil q_k \rceil)$. Such "extended" branches are computed from proofs of unsatisfiability of particular systems of Diophantine equations, which in [5] are generated by computing Hermite Normal Forms [12]. In our solver, instead, we can reuse the module for handling Diophantine equations, thanks to its proof-production capability. This makes the implementation very simple.

For lack of space, we can not provide the full details of the algorithm, and we refer to [5] for more information. Here we only mention that, as already observed in [5], 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 ongoing and future work.

Despite the fact that also this method is incomplete unless bounds for all variables are determined a priori, in [5] it was shown to be much more effective than standard branch and bound in practice.

5 Experiments

We have implemented the $\mathcal{LA}(\mathbb{Z})$-solver presented here within our new SMT solver MATHSAT5. In this section, we experimentally evaluate 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.

Description of the benchmark instances. The SMT-LIB [3] would have been a natural source of benchmarks for our evaluation, since it contains over 3900 problems in the QF_LIA (quantifier-free $\mathcal{LA}(\mathbb{Z})$) logic. Unfortunately however, from our experience almost all of those problems require virtually no reasoning over the integers: the unsatisfiable ones are unsatisfiable also in $\mathcal{LA}(\mathbb{Q})$, and for the satisfiable ones the first model found by the $\mathcal{LA}(\mathbb{Q})$-solver is already a $\mathcal{LA}(\mathbb{Z})$-model. Notable exceptions are the instances in the `rings` family, which do require integer reasoning. However, such instances can be greatly simplified by applying some preprocessing techniques on term-level if-then-else constructs [10].

Therefore, we have decided to use the following families of benchmarks, not (yet) included in the SMT-LIB:

BV have been obtained by encoding in SMT($\mathcal{LA}(\mathbb{Z})$) some bit-vector formulas from the SMT-LIB, using the encoding of [3].[4] Most of the instances are satisfiable.

CAV09 are the randomly-generated conjunctions of $\mathcal{LA}(\mathbb{Z})$-inequalities which were used in [5]. Most of the instances are satisfiable.

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

RINGS are preprocessed 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, [5] in order to prevent the application of the preprocessing technique of [10]. All the instances are unsatisfiable.

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 [10] (winner of the 2009 SMT-COMP competition on QF_LIA), Z3 [4] (winner of 2008 [6]) and YICES2 (the new version of the popular YICES solver [6]). [7] The results are collected in Figure 2. The plots show the accumulated time (on the X axis) for solving

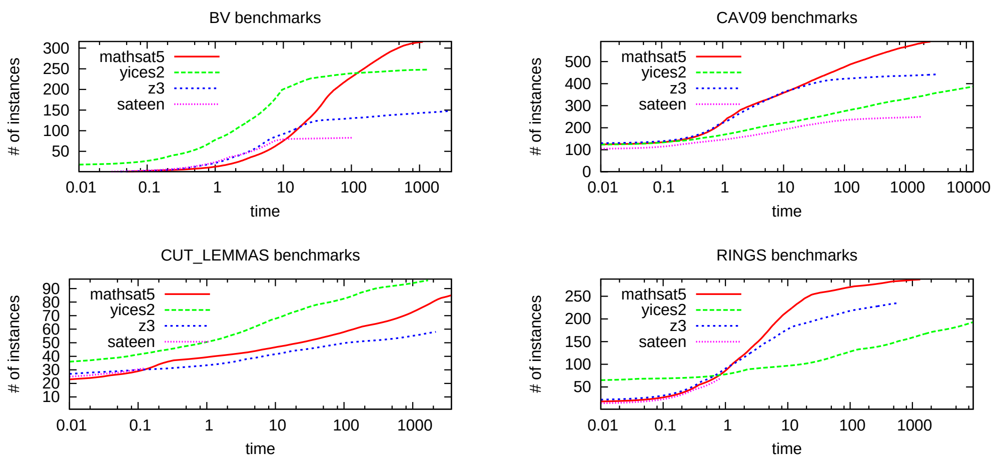
[3]`http://smtlib.org`

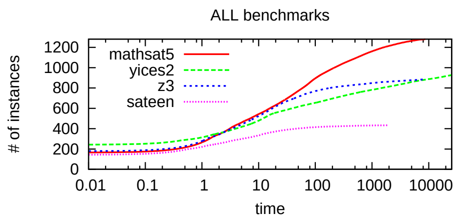
[4]The non-easily-linearizable operators have been replaced with fresh variables.

[5]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.

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

[7]We would have liked to compare also with the tool of [5], but it was not possible to obtain it.



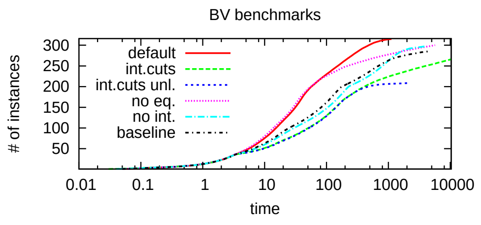


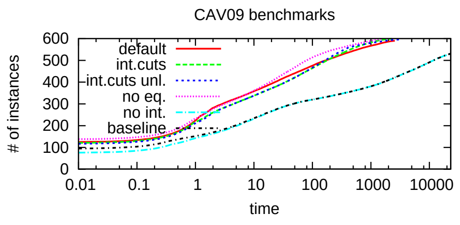
Solver	# solved	total time
MATHSAT5	1281 / 1312	8677.80
YICES2	928 / 1312	25421.15
Z3	887 / 1312	8983.43
SATEEN	435 / 1312	1908.38

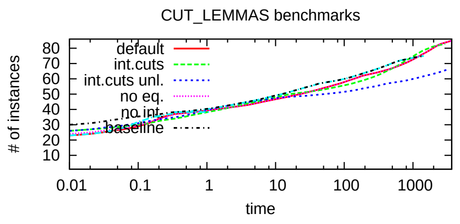
Figure 2: Comparison among different SMT solvers.

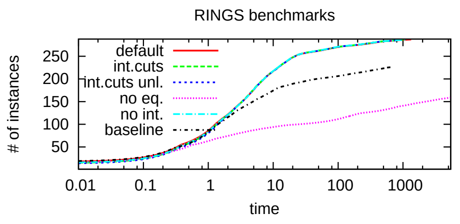
a given number of instances (on the Y axis) within the timeout, both for each individual benchmark family as well as for all the benchmarks. They show that, with the exception of the CUT_LEMMAS family on which YICES2 has very good performances, MATHSAT5 outperforms the other solvers: overall, MATHSAT5 can solve about 38% more problems than the closest competitor YICES2, with a significantly shorter total execution time. We remark that MATHSAT5 implements the same $\mathcal{LA}(\mathbb{Q})$-procedure of [6] as Z3 and YICES2, and its performance on SMT($\mathcal{LA}(\mathbb{Q})$) is comparable to that of these two solvers.

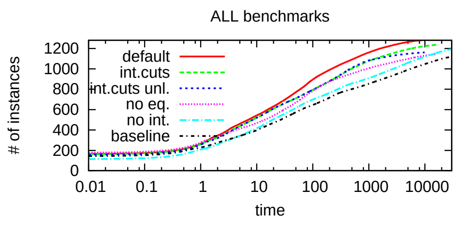
In the second part of our experiments, we compare different configurations of MATHSAT5, in order to evaluate the effectiveness of our techniques and heuristics. The results are shown in Figure 3. We ran MATHSAT5 with equality elimination and tightening disabled ('no eq.'), with the internal branch and bound disabled ('no int.'), with both disabled ('baseline'), using the cuts from proofs technique of [5] also in the internal branch 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 [5] more eagerly allows to solve 9 more instances (out of 600). However, this worsens performance in general, especially











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

Figure 3: Comparison among different $\mathcal{LA}(\mathbb{Z})$-heuristics in MATHSAT5.

on the instances of the BV family.

6 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*, volume 4246 of *LNCS*, pages 512–526. Springer, 2006.

[3] 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.

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

[5] 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*, volume 5643 of *LNCS*. Springer, 2009.

[6] 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*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

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

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

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

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

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

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

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