SPASS-SATT
A CDCL(LA) Solver

Martin Bromberger[1,2], Mathias Fleury[1,2], Simon Schwarz[1,2], and Christoph Weidenbach[1]

[1] Max Planck Institute for Informatics and Saarland University, Saarland Informatics Campus, Germany {mbromber,mfleury,sschwarz,weidenb}@mpi-inf.mpg.de

[2] Graduate School of Computer Science, Saarland Informatics Campus, Germany

Abstract. SPASS-SATT is a CDCL(LA) solver for linear rational and linear mixed/integer arithmetic. This system description explains its specific features: fast cube tests for integer solvability, bounding transformations for unbounded problems, close interaction between the SAT solver and the theory solver, efficient data structures, and small-clause-normal-form generation. SPASS-SATT is currently one of the strongest systems on the respective SMT-LIB benchmarks.
This paper has been published at CADE 27 [8].

Keywords: Linear Arithmetic, Integer Arithmetic, SMT, Preprocessing

1 Introduction

SPASS-SATT (v1.1) is a sound and complete CDCL(LA) solver for quantifier-free linear rational and linear mixed/integer arithmetic. It is a from-scratch implementation except for some basic data structures taken from the SPASS [33] superposition theorem prover. It is available through the SPASS-Workbench [3]. We participated with SPASS-SATT in the main track of the 13th International Satisfiability Modulo Theories Competition (SMT-COMP 2018) and ranked first in the category QF_LIA (quantifier-free linear integer arithmetic) [1] and second in the category QF_LRA (quantifier-free linear rational arithmetic) [2]. This system description explains the main features that led to the success of SPASS-SATT. We do not only describe the relevant techniques, but also show their specific impact on dedicated groups of examples from the SMT-LIB by experiments.

By far not all techniques presented in this system description are unique features of SPASS-SATT. The techniques that appeared first in SPASS-SATT are the unit cube test and bounding transformations explained in Section 2. Concerning preprocessing, SPASS-SATT is the first SMT solver implementing the small-clause-normal-form algorithm, see Section 4. Further important techniques implemented in SPASS-SATT have already been available in other SMT solvers such as CVC4 [4], MathSAT [13], Yices [17], and Z3 [15], but not all in one tool: (i) the implementation of branch and bound as a separate theory solver and a

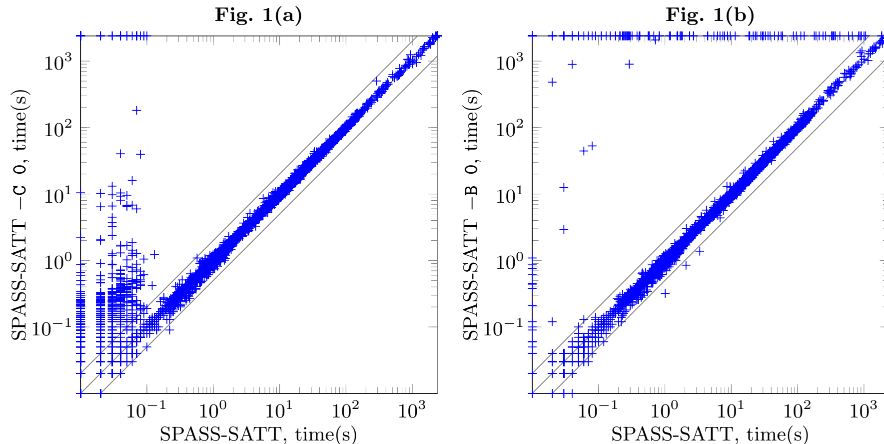


Fig. 1. Impact of our BnB extensions on the QF LIA benchmarks
(a): QF_LIA with(out) Unit Cube Tests **(b):** QF_LIA with(out) Bounding Transf.

number of improvements to the simplex implementation such as a priority queue for pivot selection, integer coefficients instead of rational coefficients, dynamically switching between native and arbitrary precision integers, and backing-up versus recalculating simplex states, all in Section 2, (*ii*) decison recommendations, unate propagations, and bound refinements for the interaction between the SAT and theory solver, in Section 3, and (*iii*) preprocessing techniques for if-then-else operators and pseudo-boolean inequalities, in Section 4. Although these techniques are contained in existing SMT solvers, not all have been described in the respective literature. The paper ends with a discussion of future extensions to SPASS-SATT in Section 5.

The benchmark experiments with SPASS-SATT consider the 6947 SMT-LIB benchmarks for quantifier-free linear integer arithmetic (QF_LIA) [5]. For the experiments, we used a Debian Linux cluster and gave SPASS-SATT for each problem one core of an Intel Xeon E5620 (2.4 GHz) processor, 8 GB RAM, and 40 minutes. The results are depicted as scatter plots and in each of them we compare the default configuration (i.e., without any command line options) of SPASS-SATT (horizontal axis) with an alternative configuration of SPASS-SATT (vertical axis). (The SMT-COMP results were obtained with the default configuration; by default all presented techniques are turned on.)

2 SPASS-IQ: An LA Theory Solver

SPASS-SATT's theory solver, called SPASS-IQ, decides conjunctions of linear arithmetic inequations. It is divided into two main components: a simplex implementation for handling linear rational arithmetic and a branch-and-bound implementation for handling linear mixed/integer arithmetic.

However, the division between the two components is in all truth not that strict. The branch-and-bound implementation is more of a supervisor for the simplex implementation. To be more precise, the branch-and-bound implementation coordinates the search for a mixed or integer solution, but the majority of the actual search/calculation is still done by the simplex implementation. For most QF_LIA benchmark instances (4894 out of 6947 instances), this supervision is not even necessary; i.e., SPASS-SATT solves these instances with just the simplex implementation as its theory solver. This means that SPASS-SATT's efficiency on the QF_LIA benchmarks also highly depends on the efficiency of our simplex implementation and not just on the extensions and optimizations to our branch-and-bound implementation.

The *simplex implementation* inside SPASS-IQ is based on a specific version [18] of the dual simplex algorithm [30]. The overall efficiency of our simplex implementation is heavily influenced by the efficiency of the data structures that we use. Our most important data structure features are:
1) *Priority Queue for Pivot Selection:* Instead of iterating over all basic variables when searching for violated basic variables, we collect the basic variables in a priority queue as soon as they become violated.
2) *Integer Coefficients Instead of Rational Coefficients:* We avoid rational coefficients in our simplex tableau by multiplying each equation in the tableau with the common denominator of the equations coefficients. As a result, each basic variable also has a coefficient, but all coefficients are integers. This transformation roughly halves the cost of most tableau operations because we do not need to consider rationals which are typically represented by two integers (the numerator and the denominator).
3) *Dynamically Switching between Native and Arbitrary-Precision Integers:* We use the *arbitrary-precision arithmetic library* FLINT to represent our integers [22]. It dynamically switches between native C integer and arbitrary-precision types.
4) *Backup vs. Recalculation:* In contrast to Dutertre and de Moura's version of the simplex algorithm, our simplex backtrack function recalculates a satisfiable assignment instead of loading a backup of the last satisfiable assignment.

SPASS-IQ's second set of decision procedures revolves around an *implementation of the branch-and-bound (BnB) algorithm* [30]. Most SMT solvers implement branch and bound through a technique called *splitting-on-demand* [6], which delegates some of the branch-and-bound reasoning to the SAT solver. In order to keep more control over the branch-and-bound reasoning, we decided against splitting-on-demand and implemented branch and bound as a theory solver separate from the SAT solver. This also made it easier to complement branch and bound with other decision procedures:

The first two extensions that we discuss here are *simple rounding* (turn off with −`LASR 0`) and *bound propagation* (turn off with −`LABP 0`) [30], which are both classical additions to most branch-and-bound implementations. For simple rounding, we round any rational solution computed during the branch-and-bound search to the closest integer assignment and check whether this is already an integer solution. For bound propagation, we propagate new bounds from ex-

isting bounds at every node in our branching tree. Although both techniques are very popular, we could only measure a minor impact on SPASS-SATT's performance on the QF_LIA benchmarks. With simple rounding we solve only one instance faster and with bound propagations we solve only 10 additional instances. In part, this is due to our next two extensions that make simple rounding and bound propagation in many cases unnecessary.

The next extension we discuss is the *unit cube test* (turn off with `−C 0`). It determines in polynomial time whether a polyhedron, i.e., the geometric representation of a system of inequalities, contains a hypercube parallel to the coordinate axes with edge length one [10,11]. The existence of such a hypercube guarantees a mixed/integer solution for the system of inequalities.

The unit cube test is only a sufficient and not a necessary test for the existence of a solution. There is at least one class of inequality systems, viz., absolutely unbounded inequality systems [10,11], where the unit cube test is also a necessary test and which are much harder for many complete decision procedures.

The plot in Figure 1(a) shows that SPASS-SATT employing the unit cube test solves 56 additional benchmark instances from the QF_LIA benchmarks and solves 705 instances more than twice as fast.[3] Moreover, the unit cube test causes only a minor overhead on problems where it is not successfully applicable.

The final extension that we discuss are *bounding transformations* (turn off with `−B 0`). Branch and bound alone is an incomplete decision procedure and only guarantees termination on bounded problems, i.e., problems where all variables have an upper and a lower bound. For this reason, we developed two transformations that reduce any unbounded problem into an equisatisfiable problem that is bounded [7]. The transformed problem can then be solved with our branch-and-bound implementation because it is complete for bounded problems.

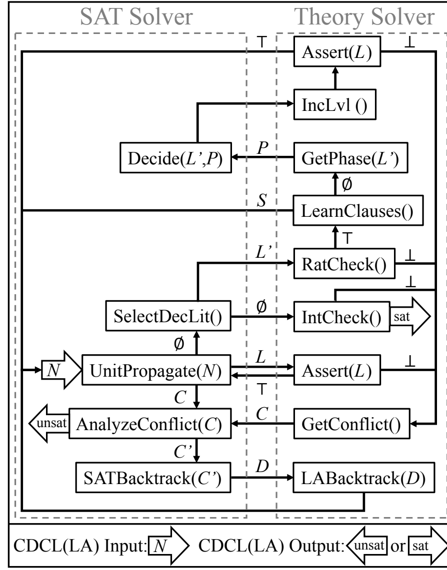
The plot in Figure 1(b) shows that SPASS-SATT employing the bounding transformation solves 169 additional benchmark instances from the QF_LIA benchmarks and solves 167 instances more than twice as fast.[4] Moreover, the bounding transformation causes only a minor, almost immeasurable overhead on problem instances where it is not successfully applicable.

3 CDCL(LA): SAT and Theory Solver Interaction

SPASS-SATT uses at its core a CDCL(LA) implementation that combines our CDCL (conflict-driven-clause-learning)-based SAT solver SPASS-SAT with our LA theory solver SPASS-IQ. The result is a decision procedure for ground linear-arithmetic formulae in clause normal form. In this section, we quickly explain how our theory solver and SAT solver interact. To this end, we list in Figure 3 the main interface functions of our SAT solver and theory solver and show through

[3] These instances belong to the `dillig` [16], `CAV-2009` [16], `slacks` [23], `20180326-Bromberger` [7], and `prime-cone` benchmark families [23], which together contain more than 1483 instances of absolutely unbounded problems.

[4] These instances belong to the `20180326-Bromberger` [7], `arctic-matrix` [14], `cut_lemmas` [20], `slacks` [23], and `tropical-matrix` [14] benchmark families.



Decide(L,P): if P is $\top$, then L is added to the model; otherwise, $\neg L$ is added.
SelectDecLit(): returns $\emptyset$ if the model satisfies all clauses; otherwise, returns a literal L that is undefined in the model.
UnitPropagate(N): returns $\emptyset$ if no literal can be unit propagated; otherwise, selects a literal L that can be unit propagated and adds it to the model; if a clause $C \in N$ evaluates to $\bot$ under the new model, then returns C; otherwise, returns L.
AnalyzeConflict(C): derives a clause C' which is the negation of the literals that led to the conflict in C; ends CDCL(LA) and returns *unsat* if C' is empty; otherwise, returns C'.
SATBacktrack(C'): adds C' to the clause set N and backtracks to the maximum decision level D where C' is still satisfiable; retuns D.
Assert(L): returns $\bot$ if the literal L contradicts another asserted literal.
IncLvl(): notifies the theory solver that a new decision level was reached.
GetPhase(L): selects the phase P for the decision literal L.
LearnClauses(): Adds a set S of clauses to N that correspond to unate propagations and bound refinements; returns $\emptyset$ if there are no clauses to learn.
RatCheck(): determines a rational solution for the asserted literals; returns $\top$ if a rational solution exists; otherwise, returns $\bot$.
IntCheck(): determines an integer solution for the asserted literals; ends CDCL(LA) and returns *sat* if an integer solution exists; otherwise, returns $\bot$.
GetConflict(): returns a clause C that explains the theory conflict.
LABacktrack(D): removes all asserted literals that were added after decision level D; recalculates a rational solution for the remaining asserted literals.

Fig. 2. CDCL(LA) as implemented in SPASS-SATT

a flow graph how they interact. The main focus of this section, however, is to explain in which way our implementation of CDCL(LA) differs from more general frameworks for CDCL(T), also called DPLL(T) [6,19,27,28].

There are three key points that we have changed compared to the more general frameworks for CDCL(T). First of all, we rely on "weakened early pruning" [31], i.e., we only use a weaker but faster check to determine theory satisfiability for partial (propositionally abstracted) models. We do so because IntCheck(), i.e., checking for an integer solution, is too expensive and not incrementally efficient enough to be checked more than once per complete (propositionally abstracted) model. As a compromise, we at least check the partial model

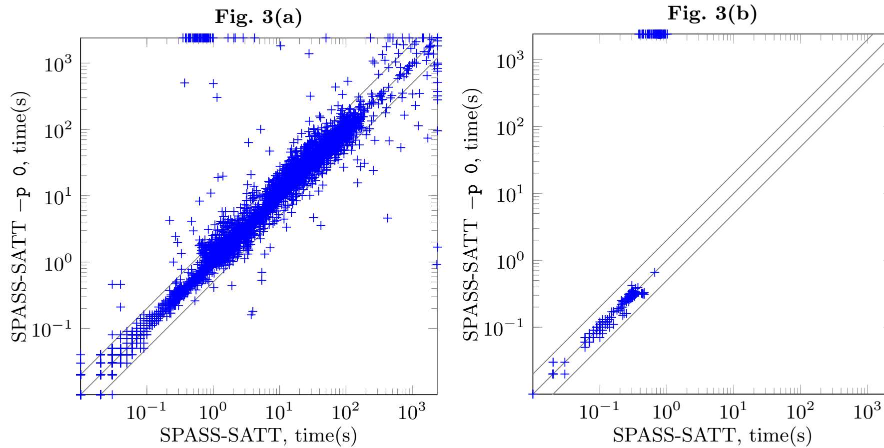


Fig. 3. Impact of decision recommendations on the QF_LIA benchmarks
(a): QF_LIA with(out) decision recom. **(b):** `convert` with(out) decision recom.

with RatCheck(), i.e., we check for a rational solution, before we add a(nother) decision literal to the model with Decide(L,P).

As our second key change, we let the theory solver select via GetPhase(L) the phase of the next decision literal L, i.e., whether Decide(L,P) will add the positive or the negated version of L to the model. We call this technique a *decision recommendation* (turn off with `-p 0`).[5] Finally, we use theory reasoning via the function LearnClauses() to find and learn new clauses implied by the input formula. The reasoning techniques we use for this purpose are *unate propagations* and *bound refinements* as proposed in [18].

In Figure 3(a), we examine the impact that decision recommendations have on SPASS-SATT's performance on the QF_LIA benchmarks. With decision recommendations it can solve 129 additional problems. Moreover, it becomes more than twice as fast on 389 problems, but only twice as slow on 58 problems. The benchmark family that is impacted the most by decision recommendations is `convert` with 116 additionally solved problems (see Figure 3(b)). Although SPASS-SATT frequently and regularly performs unate propagations on the QF_LIA benchmarks, we are unable to observe any consistent benefit or drawback from this interaction technique. The impact of bound refinements is also relatively minor and SPASS-SATT solves only 24 additional problem instances if they are activated.

[5] In our own theory solver, GetPhase(L)=$\top$ if the current theory assignment satisfies L or if Assert($\neg L$) would return $\bot$. Otherwise, GetPhase(L)=$\bot$. (If unate propagations are enabled, then the case that Assert($\neg L$) would return $\bot$ is impossible because L would have been unate propagated.)

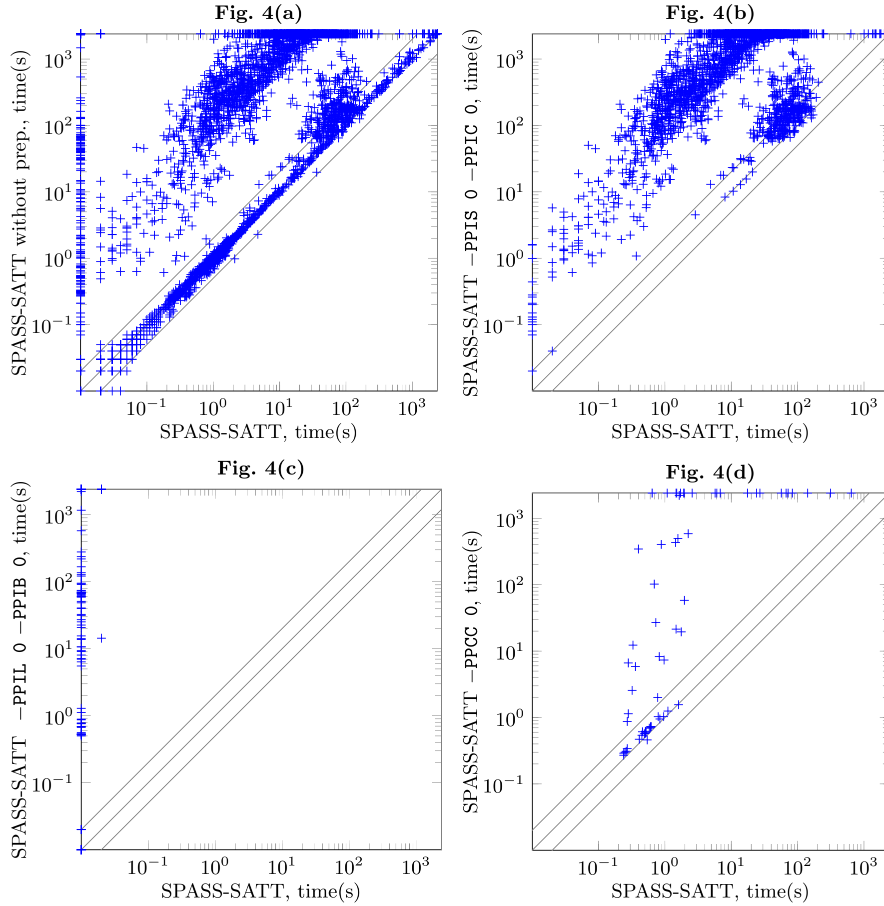


Fig. 4. Impact of our preprocessing techniques on **(a)** the QF_LIA benchmarks and more specifically **(b)** `nec_smt`, **(c)** `rings`, and **(d)** `pb2010`

4 Preprocessing

Many real world applications can be encoded as linear integer arithmetic formulas, and some of those applications are too specialized to be efficiently handled by our rather general CDCL(LA) implementation. To resolve this, we have complemented SPASS-SATT with several specialized preprocessing techniques.[6]

Complex input formulas are typically transformed into CNF by a Tseitin-style renaming [32] using a static criterion which subformula to replace by a fresh propositional variable. SPASS-SATT includes the small clause normal form

[6] All preprocessing techniques are also contained in CVC4 [4] with the exception of the small CNF. The implementation is so efficient because we employ a shared term representation and cache all intermediate results.

algorithm [29]. Instead of a static criterion, the number of clauses with or without a renaming is compared and a fresh propositional variable is introduced only if a renaming eventually yields fewer clauses. This results in a more compact CNF with strictly fewer additional propositional variables. To this end we extended the small clause normal form algorithm to ITE formulas. For an ITE formula (ite t_1 t_2 t_3), simplified by the below techniques and potentially contained in some formula f, we compare the number of clauses generated out of replacing the formula with a fresh variable P in f and adding $P \leftrightarrow [(t_1 \rightarrow t_2) \wedge (\neg t_1 \rightarrow t_3)]$ with a direct replacement of the ITE formula by the before mentioned conjunction of two implications. This test can be carried out in constant time after having once filled respective data structures. The set up of the data structures needs one run on the overall formula, i.e., can be computed in linear time [29].

SPASS-SATT also has five specialized preprocessing techniques for *if-then-else expressions* (ITE). Our first technique, *if-then-else reconstruction* (turn off with −`PPIR 0`), rebuilds if-then-else operations that were already preprocessed-away by the creators of the input problem. The reconstruction then allows us to apply simplifications missed during the creation of the input problem. To this end, we check whether the first conjunctive layer of our formula $f :=$ (and $\ldots$ t_i $\ldots$ t'_i $\ldots$) contains any pair of clauses t_i, t'_i that match the clauses added by the standard if-then-else elimination, i.e., $t_i \in T_i$ and $t'_i \in T_i \setminus \{t_i\}$, where $T_i = \{(\text{or } t_{i1} \ (= \ y_i \ t_{i3})), (\text{or } (\text{not } t_{i1}) \ (= \ y_i \ t_{i2})), (\text{or } (= \ y_i \ t_{i2}) \ (= \ y_i \ t_{i3}))\}$ and y_i is an arithmetic variable. If we find such pairs t_i, t'_i, then we remove them from f and replace all remaining occurrences of y_i in f with (ite t_{i1} t_{i2} t_{i3}).

The next three techniques are all dedicated to so-called *constant if-then-else expressions* (CITEs). A CITE is either a leaf, i.e., an arithmetic expression that can be simplified to a number $a_{ij} \in \mathbb{Q}$, or a branch, i.e., an if-then-else expression (ite t_1 t_2 t_3) where t_2 and t_3 are again CITEs.

The first CITE technique is called *shared monomial lifting* (turn off with −`PPIL 0`) and we use it to increase the number of CITEs in our formula. It traverses the subterms in our formula in bottom-up order and transforms all subterms $t :=$ (ite t_1 $(+ \ q \ q')$ $(+ \ q \ \hat{q})$) into $(+ \ q \ (* \ 1 \ (\text{ite } t_1 \ (+ \ q') \ (+ \ \hat{q}))))$. (We assume here for simplicity that the shared part q appears after the unshared parts q' and $\hat{q}$. In reality SPASS-SATT has to find and extract the shared parts.)

The second technique is called *CITE simplification* (turn off with −`PPIS 0`) and it simplifies atoms (o t_1 t_2), where t_1 and t_2 are CITEs and o is one of the operators <=, =, >=. To be more precise, the technique essentially pushes the comparison operator o recursively down the CITE branches and greedily simplifies any branch to *true* or *false* if possible. For more details, see Section 4 in the paper by Kim et al. on efficient term-ITE conversion [25].

The third technique is called *CITE bounding* (turn off with −`PPIB 0`) and eliminates the remaining CITEs. Thanks to it, we only introduce one new variable for each topmost CITE instead of one variable for each CITE branch. Moreover, this new variable describes only a small set of values (often equivalent to just the CITE leaves) because it is bounded as tightly as possible.

As its first step, CITE bounding creates one new integer variable x_j for each topmost CITE expression t_j and replaces the occurrences of t_j in our formula f with x_j. As its second step, CITE bounding extends f to the formula $f' :=$ (and f f_1 f'_1 $\ldots$ f_m f'_m), where (*i*) f_j is equivalent to t_j except that all leaves a_{ij} of t_j are replaced by the equations (= x_j a_{ij}), (*ii*) $f'_j :=$ (and (>= x_j a_{minj}) (<= x_j a_{maxj})), and (*iii*) a_{minj} is the smallest leaf in t_j and a_{maxj} is the largest leaf in t_j. As its last step, CITE bounding replaces all occurrences of x_j in f' with ($*$ a_{gj} x_j), where a_{gj} is the greatest common divisor of the leaves a_{ij} in t_j.

The final ITE technique handles *nested conjunctive if-then-else expressions* (AND-ITEs). An AND-ITE is a series of nested if-then-else expressions that can be simplified to a conjunction. For instance, if $t_i :=$ (ite t'_i t_{i+1} *false*) for $i = 1, \ldots, n$, then t_1 is an AND-ITE equivalent to (and t'_1 $\ldots$ t'_n t_{n+1}). Naturally, we transform these AND-ITEs into actual conjunctions and we call this process compression. However, we compress an AND-ITE t_1 only if all of its actual AND-ITE subterms t_i appear only inside t_1 and only once. If this is the case, then we first replace all occurrences of t_1 in f by a new propositional variable p_j and extend our formula f to the formula (and f (= p_j t'_1)), where t'_1 is the compressed form of t_1. (If this is not the case, then we simply replace and compress the AND-ITE subterms in t_i first.) We do the compression in this way to strengthen the connection of the AND-ITEs that have multiple occurrences in f. The above described technique is called *if-then-else compression* (turn off with −`PPIC 0`) and it was first presented by Burch in [12]. However, Burch used it to simplify control circuits and not SMT input problems.

Last but not least, SPASS-SATT also provides a preprocessing technique for *pseudo-boolean problems* [21], i.e., linear arithmetic problems where all integer variables x_j have bounds $0 \leq x_j \leq 1$, which we call *pseudo-boolean variables*. To be more precise, SPASS-SATT recognizes clauses that are encoded as linear pseudo-boolean inequalities (i.e., inequalities containing just pseudo-boolean variables) and turns them into actual clauses (turn off with −`PPCC 0`). (This technique goes back to the NP-hardness proof of 0-1 programming [24].) However, SPASS-SATT only transforms inequalities containing at most three variables because it would otherwise fail to solve some of the problems from the `pidgeons` benchmark family.

The `convert` benchmark family contains problems that are relatively hard unless SPASS-SATT uses the right combination of techniques. To be more precise, SPASS-SATT solves only two-thirds of the 319 instances if one of the following three techniques is missing: (*i*) recalculating simplex states during backtracking (Section 2), (*ii*) decision recommendations (Section 3), or (*iii*) the small CNF transformation. SPASS-SATT with all three techniques solves all instances in less than 2 seconds (see Figure 3(b)).

The `nec_smt` benchmark family contains problems with many nested if-then-else and let expressions. SPASS-SATT can handle most of them if we first apply our constant if-then-else simplifications and our conjunctive if-then-else compression. In Figure 4(b), we see that SPASS-SATT without our preprocessing

techniques solves only 1422 out of the 2800 benchmark instances and is by far slower on the instances it can solve. SPASS-SATT with our preprocessing techniques solves 2782 out of the 2800 benchmark instances.

The `rings` benchmark family encodes associative properties on modular arithmetic with the help of if-then-else expressions. With a combination of shared monomial lifting and constant if-then-else bounding, these problems become almost trivial to solve. In fact, SPASS-SATT needs less than one second for each problem instance and needs only techniques for linear rational arithmetic to solve each of them (Figure 4(c)).

The `rings_preprocessed` benchmark family is equivalent to the `rings` benchmark family except that all if-then-else-operations were eliminated by standard if-then-else elimination [20]. We can use the same trick as for the `rings` benchmark family if we first use our if-then-else reconstruction technique that reverses the standard if-then-else elimination.

The `pb2010` benchmark family is a set of industrial problems taken from the pseudo-boolean competition 2010. With its pseudo-boolean preprocessing, SPASS-SATT solves 22 additional benchmark instances from the 81 instances in the `pb2010` benchmark family (see Figure 4(e)). Moreover, SPASS-SATT solves all of these instances without its branch-and-bound implementation.

5 Conclusion and Future Work

We have presented SPASS-SATT our complete solver for ground linear arithmetic and have explained which techniques make it so efficient in practice. To summarize, SPASS-SATT is so efficient because (*i*) we have optimized the data structures in our simplex implementation, (*ii*) we have combined branch and bound with the unit cube test and the bounding transformation, (*iii*) we have added decision recommendations to our CDCL(LA) framework, (*iv*) we have added a small CNF transformation, and (*v*) we have added specialized preprocessing techniques for if-then-else expressions and pseudo-boolean inequalities.

Almost all of the presented techniques can be applied incrementally, however this is not always useful. For the partial models computed by the SAT solver, we only apply the simplex method, unate propagation, and bound refinements incrementally. If we ever extend SPASS-SATT to handle theory combinations or incremental SMT-LIB problems, then we would also apply branch-and-bound, unit cubes, and bounding transformations incrementally; but only on the models generated during Nelson-Oppen combination or between two (check-sat) calls.

For future research, we plan to extend SPASS-SATT to quantified linear arithmetic. Moreover, we plan to complement SPASS-SATT with several specialized decision procedures. For instance, SPASS-SATT could handle (almost) pseudo-boolean problems (e.g., benchmark families `pb2010`, `miplib2003`) much more efficiently if we extended branch and bound with a SAT based arithmetic decision procedure [9,23,26].

References

1. SMT-COMP 2018 results for QF_LIA (main track). `http://smtcomp.sourceforge.net/2018/results-QF_LIA.shtml`.
2. SMT-COMP 2018 results for QF_LRA (main track). `http://smtcomp.sourceforge.net/2018/results-QF_LRA.shtml`.
3. The SPASS Workbench. `https://www.spass-prover.org/`.
4. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*. 2011.
5. C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
6. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT modulo theories. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *LNCS*, pages 512–526. 2006.
7. M. Bromberger. A reduction from unbounded linear mixed arithmetic problems into bounded problems. In *Automated Reasoning*, volume 10900 of *LNCS*, 2018.
8. M. Bromberger, M. Fleury, S. Schwarz, and C. Weidenbach. SPASS-SATT - A CDCL(LA) solver. In P. Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2019.
9. M. Bromberger, T. Sturm, and C. Weidenbach. Linear integer arithmetic revisited. In *CADE-25*, volume 9195 of *LNCS*. 2015.
10. M. Bromberger and C. Weidenbach. Fast cube tests for LIA constraint solving. In *IJCAR 2016*, volume 9706 of *LNCS*. 2016.
11. M. Bromberger and C. Weidenbach. New Techniques for Linear Arithmetic: Cubes and Equalities. *Formal Methods in System Design*, 51(3), 2017.
12. J. R. Burch. Techniques for verifying superscalar microprocessors. In *DAC*, pages 552–557. ACM Press, 1996.
13. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*, 2013.
14. M. Codish, Y. Fekete, C. Fuhs, J. Giesl, and J. Waldmann. Exotic semi-ring constraints. In *SMT@IJCAR*, volume 20 of *EPiC Series in Computing*, pages 88–97. EasyChair, 2012.
15. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
16. I. Dillig, T. Dillig, and A. Aiken. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In *CAV*, volume 5643 of *LNCS*. 2009.
17. B. Dutertre. Yices 2.2. In *CAV 2014*, volume 8559 of *LNCS*, 2014.
18. B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of *LNCS*. 2006. Extended version: Integrating simplex with DPLL(T). Tech. rep., CSL, SRI INTERNATIONAL (2006).
19. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): fast decision procedures. In *CAV*. Springer, 2004.
20. A. Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *JSAT*, 8(1/2), 2012.
21. P. L. Hammer and S. Rudeanu. *Boolean methods in operations research and related areas*, volume 7 of *Econometrics and Operations Research*. Springer Science & Business Media, 2012.

22. W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*, 2013. Version 2.4.0, `http://flintlib.org`.
23. D. Jovanović and L. M. de Moura. Cutting to the chase solving linear integer arithmetic. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2011.
24. R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
25. H. Kim, F. Somenzi, and H. Jin. Efficient term-ite conversion for satisfiability modulo theories. In *SAT*, volume 5584 of *LNCS*, pages 195–208. Springer, 2009.
26. R. Nieuwenhuis. The IntSat method for integer linear programming. In *Principles and Practice of Constraint Programming*, volume 8656 of *LNCS*. 2014.
27. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *CAV*, pages 321–334. Springer, 2005.
28. 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.
29. A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning*, volume 1. Elsevier, 2001.
30. A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1986.
31. R. Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.
32. G. Tseitin. On the complexity of derivations in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers on Computational Logic*, volume 2, pages 466–483. Springer, 1983. First Published in: *Studies in Constructive Mathematics and Mathematical Logic*, (Slisenko, A.O., ed.), 1968.
33. C. Weidenbach, D. Dimova, A. Fietzke, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145, Montreal, Canada, August 2009. Springer.