

A Novel SAT Procedure for Linear Real Arithmetic

Peter Koppensteiner
nfotex ITDL GmbH, Vienna, Austria
peter.koppensteiner@nfotex.com

Helmut Veith
Institut für Informatik, TU München
veith@in.tum.de

Abstract

The satisfiability of linear arithmetic logic is an important question with applications in verification and constraint satisfaction, and has been studied intensively during the last years. Most existing procedures attempt to decouple the SAT solver and the mathematical solver; this decoupled approach has the advantage that state-of-the-art solvers can be employed, but the blindness of the SAT solver with respect to arithmetic content requires subtle protocols to exploit the output of the mathsolver in the SAT DPLL procedure. In this paper, we propose to integrate mathematical constraint solving capabilities directly into the SAT solver. We introduce a new solver for linear arithmetic logic which is based on the classical SAT paradigms, but treats the SAT literals according to their arithmetic interpretation, employing stepwise Fourier-Motzkin elimination on the fly during the DPLL search. Characteristically, our solver introduces new literals which correspond to auxiliary mathematical formulas, and new clauses which enable the SAT solver to treat the arithmetic proofs as integral parts of the SAT proofs. The conflict clauses in our approach immediately lead to arithmetic explanations, and all required backtracking is directly achieved by the SAT solver. Moreover, the new literals do not increase the complexity of SAT solving, but amount to a form of unit propagation. We believe that our approach is not limited to linear arithmetic but applicable to a wide class of problems where elimination methods in the style of Fourier-Motzkin are available. Our results are substantiated by positive experiences with a prototype implementation for linear arithmetic with $=$ and $<$.

1 Introduction

Linear real number constraints are a natural formalism to describe a large number of practically important problems in areas such as scheduling, optimization and verification. In contrast to linear programming which deals with conjunctions of inequalities, we are interested in algorithms for arbitrary Boolean combinations of inequalities, i.e., quantifier-free formulas whose atomic predicates are linear inequalities. Consider for example the formula $(\mathbf{a} \leq 1.1 \vee \mathbf{a} = 2) \wedge (\mathbf{b} = 1 \vee \mathbf{b} = 2.3) \wedge (\mathbf{a} + \mathbf{b} < 3 \vee 2\mathbf{a} + \mathbf{b} = .7)$ which has two free first-order variables, and is in fact satisfiable. Since the problem instance is quantifier-free, and thus very similar to a SAT instance, it is natural to use SAT algorithms for solving linear constraints [2]. In fact, several promising approaches which combine SAT solvers with mathematical solvers (“mathsolvers”, e.g. Linear Programming tools) have been presented in the literature. These tools include MathSAT [2, 8] as well as more general theorem provers/decision procedures such as CVC [5] and ICS [14]. Other tools such as UCLID [20], BFM [25], and ZAPATO [3], and HDPLL [22, 17] in contrast decide linear *integer* constraints.

The common idea in the mentioned approaches is to have the SAT solver and the mathematical solver interact in accordance with a certain protocol. In the simplest case, the SAT solver will enumerate all Boolean solutions of the Boolean “skeleton” $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6)$ of the formula. Every solution is a conjunction of literals, and can therefore be viewed as a system of inequalities which are solved by a mathsolver. Our example already shows the main drawback of this approach: the Boolean abstraction usually ignores the mathematical relationships between the subformulas represented by the x_i , and will therefore most likely enumerate many Boolean solutions which have to be discarded by the mathematical solver. Thus,

this strategy would fit best in cases unlike the one presented above where the pure Boolean dependencies strongly outnumber the mathematical dependencies. While the mentioned tools of course use much more subtle approaches, cf. Section 3, they are all essentially based on a software architecture which decouples the SAT solver from the mathsolver. This has the big advantage that highly optimized SAT solvers can be used, but puts a large number of non-trivial requirements on the mathsolver [15]. The SAT solver itself has no insight into the semantic meaning of the propositional formulas – for example, it will treat $\mathbf{a} = 0$ and $\mathbf{a} < 0$ as totally unrelated Boolean propositions x_1, x_2 . In the decoupled approach, the architecture of the tools predetermines the solving strategy to a quite large extent. This is in contrast to the classical SAT solvers which owe their stable excellent performance to the ease of applying different solving strategies and heuristics, e.g., in decision ordering.

In fact, today’s SAT solvers have reached consistently high performance and a maturity reflected in a common compact architecture. Starting with GRASP [24], all modern SAT solvers employ DPLL [9] with non-chronological backtracking based on conflict clauses. The mechanics of SAT solvers was further significantly improved by CHAFF [21] which demonstrated how to perform backtracking with minimal costs. The different SAT solvers can be distinguished mainly by their strategies to choose literals, to perform restarts, and to keep the clause database small. Examples of such SAT solvers include GRASP [24], CHAFF [21], Limmat [6], BerkMin [16], and the highly compact MiniSAT [13] etc. We will refer to this group as “canonical SAT solvers” further on.

In this paper, we address the question how the algorithmics of canonical SAT solvers can be used for solving mathematical constraints in a direct manner: We present a compact solver which inherits the architecture and the algorithmic advantages of canonical SAT solvers, but is able to deal with arithmetic constraints directly and efficiently. The key idea is to enable the SAT solver to *interpret* the mathematical meaning of literals as inherent part of Boolean constraint propagation. More precisely, our SAT solver deduces *all mathematically required* consequences from true literals, and to introduce new literals and clauses which express the mathematical deduction in the SAT solving process. Note that the SAT solver does not deduce *all* (infinitely many) consequences, but only a finite set of consequences, which, by the completeness of the Fourier-Motzkin elimination, suffices to prove all contradictions. To achieve this effect, the subroutine for mathematical reasoning writes down all proof steps into the clause database in the same way the SAT solver would do, *if the mathematical information had been provided by the user as an additional input clause*. Consequently, the full power of SAT solving becomes available to solve mathematical constraints.

When, for example, the SAT solver interprets literals x_1, x_2 standing for $\mathbf{a} = 0$ and $\mathbf{a} < 0$, it will recognize a contradiction $0 < 0$, and invoke the usual non-chronological backtracking mechanism. On the other hand, when the SAT solver interprets literals x_3, x_4 standing for e.g. $\mathbf{a} + \mathbf{b} < 3$ and $-\mathbf{a} \leq 2$, it will infer $\mathbf{b} < 5$, introduce a new literal $l_{\mathbf{b} < 5}$ which represents $\mathbf{b} < 5$, and add the formula $x_3 \wedge x_4 \rightarrow l_{\mathbf{b} < 5}$, i.e., the clause $\neg x_3 \vee \neg x_4 \vee l_{\mathbf{b} < 5}$, to the clause database. Note that in contrast to the approach of ICS in [10], we do not only *a posteriori* add clauses to explain conflicts found by the mathsolver, but actively add mathematical information to the clause database, cf. Section 3 for a more detailed discussion.

In our solver, the mathematical power to reason about inequalities is mainly provided by Fourier-Motzkin elimination [19], a classic approach to verify the consistency of linear inequalities. While Fourier-Motzkin has the reputation to be slow for linear programming, it lends itself naturally for our approach (and is also used internally in ICS and CVC), since it is based on a simple elementary step, namely, the linear combination of two inequalities. As a second example and further improvement of our method we show how to deal with equalities in a similar way, using a simple version of Shostak equality propagation [19]. In summary, we believe that our method has new features and potential advantages in comparison to decoupled approaches:

- The backtracking mechanism of the SAT solver is *directly* used for mathematical backtracking, and the conflict clause calculation is directly used for finding mathematical explanations of contradictions. Consequently, the solver in essence learns mathematical properties.

- In case of unsatisfiability, the solver can output a mathematical unsatisfiable core; in fact, the mathematical unsatisfiable core coincides with the Boolean unsatisfiable core of the extended clause database.
- The solver is very compact, and requires simple changes to existing SAT solvers. The mathematical subroutines for the manipulation and interpretation of inequalities amount to a few lines of code.
- The architecture of the solver provides a simple and intuitive way to integrate mathematical reasoning tightly into SAT solvers. By varying the decision order of the SAT solver, we can simulate the principal strategies of other tools, in particular switch between “eager” and “lazy” approaches. Our arguments are substantiated by first successful experiments with a prototype solver.

Due to space restrictions, the current paper focuses on the basic idea of integrating Fourier-Motzkin elimination into a SAT solver; the equality propagation, the implementation and the experimental results are described in the full version [18] of the paper.

2 Technical Preliminaries

Linear Arithmetic Formulas and Boolean Abstractions. Our constraint language LAL contains propositional variables as in SAT as well as quantifier-free first-order formulas built from relational symbols $<$, \leq and $=$ and the operation $+$. It is easy to see that $<$ can define \leq and $=$. Thus, we will first describe an algorithm which deals with pure inequalities, and then show to make it more efficient by handling equality directly. A term is an expression built from addition $+$, constants, and first-order variables which are usually typeset in boldface. A substitution is an expression of the form $\mathbf{v} := t$ where t is a term and \mathbf{v} is a first order variable which does not occur in t . When we apply a substitution $\mathbf{v} := t$ to a formula ϕ , we replace all occurrences of \mathbf{v} in ϕ by t . Note that we will use $=$ as relational symbol in first order logic, and $:=$ to denote substitutions. To avoid confusion, we will write \simeq to denote syntactic (meta-)equality, e.g., we may write $\phi \simeq (\mathbf{a} = 5)$ to denote that ϕ is the formula $(\mathbf{a} = 5)$.

A constraint ϕ is an atomic first-order formula or its negation, i.e., a first-order literal. We write $\text{type}(\phi)$ to denote the main relation symbol in ϕ , i.e., if ϕ has the format $t_1 < t_2$ or $t_1 \leq t_2$ then $\text{type}(x) = T_<$, if ϕ has the format $t_1 = t_2$, then $\text{type}(x) = T_=$, and if ϕ has the format $\neg(t_1 = t_2)$, then $\text{type}(x) = T_{\neq}$.

Recall from the introduction that our SAT solver will use literals of different formats: some of them are ordinary propositional literals, and others will be “interpreted”, i.e., describe constraints. We will tacitly assume that the input is automatically translated into a propositional formula, as in the example from the introduction involving $(\mathbf{a} \leq 1 \vee \mathbf{a} = 2) \wedge (\mathbf{b} = 1 \vee \mathbf{b} = 2) \wedge (\mathbf{a} + \mathbf{b} < 3 \vee 2\mathbf{a} + \mathbf{b} = 6)$ and $(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (x_5 \vee x_6)$. In order to switch easily between the two abstraction levels, we write $\text{formula}(x_1) \simeq \mathbf{a} \leq 1$, $\text{formula}(x_3 \vee x_4) \simeq (\mathbf{b} = 1 \vee \mathbf{b} = 2)$, etc. Thus, $\text{formula}()$ *syntactically* translates propositional formulas to LAL formulas in the natural way. At some point in the algorithm we will need to introduce new literals which represent given constraints. We will write l_ϕ to denote a new literal, s.t. $\text{formula}(l_\phi) = \phi$.

We assume that the reader is familiar with the basic principles of canonical SAT solvers. We will write $\text{value}(x)$ to denote the truth value which the SAT solver has assigned to proposition x . Note that value is in principle three-valued, i.e., it can take the values $\{\text{true}, \text{false}, \perp\}$, where \perp means that the value is undefined. value is also naturally extended to literals and propositional formulas. For example for a clause C , $\text{value}(C) = \text{true}$ if the (partial) assignment by the SAT solver makes C true.

We use a special notation to describe the interpretation of mathematical literals: We write $[[x]]$ to denote either $\text{formula}(x)$ or $\neg\text{formula}(x)$, *depending on the value of* $\text{value}(x)$. For example, if $\text{formula}(x) \simeq \mathbf{a} > 1$, and $\text{value}(x) = \text{false}$, then $[[x]] \simeq \mathbf{a} \leq 1$, and if $\text{formula}(y) \simeq \mathbf{a} \neq 7$, and if $\text{value}(y) = \text{true}$, then $[[y]] \simeq \mathbf{a} \neq 7$. Similarly, if $\text{value}(y) = \text{false}$, then $[[y]] \simeq (\mathbf{a} = 7)$. When the meaning is clear, we will also write $\text{type}(x)$ instead of $\text{type}([[x]])$.

Fourier-Motzkin Elimination. The Fourier-Motzkin elimination method is a decision procedure for systems of linear inequalities which is based on the following simple idea: Suppose we have a linear inequality $\phi \simeq \sum_{1 \leq i \leq n} c_i \mathbf{x}_i \leq c_0$ and we choose one of the variables \mathbf{x}_1 without loss of generality. Then, depending on whether $c_1 > 0$, we can rewrite ϕ either in the form $\mathbf{x}_1 \leq f(\mathbf{x}_2, \dots, \mathbf{x}_n)$ or in the form $g(\mathbf{x}_2, \dots, \mathbf{x}_n) \leq \mathbf{x}_1$, where f and g are linear functions; in the first case, f is an upper bound u for \mathbf{x}_1 , and in the second case, g a lower bound l for \mathbf{x}_1 . Suppose now that we have two inequalities ϕ_1 and ϕ_2 which give us lower and upper bounds $l(\mathbf{x}_2, \dots, \mathbf{x}_n) \leq \mathbf{x}_1 \leq u(\mathbf{x}_2, \dots, \mathbf{x}_n)$. Then we can introduce the new constraint $l(\mathbf{x}_2, \dots, \mathbf{x}_n) \leq u(\mathbf{x}_2, \dots, \mathbf{x}_n)$ which eliminates the variable \mathbf{x}_1 . In this case, we say that the Fourier-Motzkin elimination is applicable for ϕ_1 and ϕ_2 (with respect to variable \mathbf{x}_1), and write $\phi_1 \oplus_{\mathbf{x}_1} \phi_2$ to denote the new constraint. For $\phi_1 \simeq \sum_{1 \leq i \leq n} a_i \mathbf{x}_i \leq a_0$ and $\phi_2 \simeq \sum_{1 \leq i \leq n} b_i \mathbf{x}_i \leq b_0$ it is easy to see that $\phi_1 \oplus_{\mathbf{x}_j} \phi_2$ is applicable with respect to variable x_j iff the signs of a_j and b_j are different. Note that the elimination is done in a similar way as in Gauss elimination: if two variables have coefficients with different signs in different equations, we can multiply one equation by a positive constant, and eliminate the variable by adding the two equations. The Fourier-Motzkin method also works for strict inequalities with $<$. If, for example, we have $l \leq \mathbf{x}_1 < u$, then the new constraint is $l < u$. Geometrically, the operation $\oplus_{\mathbf{x}_i}$ amounts to a projection on the \mathbf{x}_i -axis.

The classical Fourier-Motzkin method works as follows: Given a set $C = \{\phi_1, \dots, \phi_m\}$ of constraints with variables $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$, it first computes all projections $\phi_i \oplus_{\mathbf{v}_1} \phi_j$, where the operation is applicable, obtaining a set $C_{\mathbf{v}_1}$ of constraints. Then it applies $\oplus_{\mathbf{v}_2}$ to all constraints in $C_{\mathbf{v}_1}$ yielding a set $C_{\mathbf{v}_1, \mathbf{v}_2}$ etc. This approach is well-known to be complete: if the constraints in C are not satisfiable, then we will sooner or later derive a trivially false constraint involving only constants, e.g. $1 < 0$. Otherwise, if no trivial contradiction is found, C is satisfiable. Note that the Fourier-Motzkin approach computes a finite number of at most n^2 new inequalities. For a systematic logical treatment of the Fourier-Motzkin method which emphasizes the relationship between solvability and logical consistency, as well as the similarity to Gaussian elimination, we refer the reader to Kuhn [19].

3 Architectures in Related Work

Among the general strategies of solvers for linear constraints, we can distinguish roughly between lazy and eager approaches: in a classical lazy approach, the tool chain starts with a SAT solver, and uses a mathsolver only for the cases which are not excluded by the SAT solver in the first step. The initial version of MathSAT [2] used a classical lazy algorithm. A classically eager approach in contrast uses the mathsolver at the beginning of the tool chain, thus totally reducing the problem to a SAT instance. Most of the newer tools however cannot easily be classified in this simple schema – they use SAT and mathsolvers in an intertwined manner. Clearly on the eager side, however, are the UCLID tool [20] which however does not handle real linear arithmetic, and the BFM tool [25] which we will describe in more detail below.

The main challenge of the lazy approach stems from the possibility that the SAT instance is underconstrained, and that the SAT solver may generate too many models which will later be discarded by the mathsolver. MathSAT alleviates this problem by a mix of strategies, including a preprocessing step which checks for pairs of constraints whether they are inconsistent and introduces new clauses which express inconsistencies, and an incremental layered mathematical solver which avoids redundant computations and selects an appropriate constraint-solving algorithm depending on the input [7].

MathSAT is also an example of the *decoupled architecture* which decouples the work of the SAT solver from the mathematical solver: from the point of view of the SAT solver, no mathematical constraints are visible at all. This architectural choice puts strong requirements on the mathsolver, and much less so on the SAT solver; moreover, the decoupled approach necessitates an often subtle protocol to translate the results of the math solver back to the SAT solver. A first systematic analysis of the interaction between the SAT solver

and the mathsolver is given in [15], where the external mathsolver is viewed as a black box, and the required behavior of the mathsolver is specified in a generic manner, independently of the underlying theory. This approach has been successfully used for theories such as uninterpreted function symbols.

The decoupled architecture is also implemented in the more general (and highly efficient) solvers ICS [14, 12, 10] and CVC [5, 26]: although their procedures differ from MathSAT, the internal SAT solver remains blind to the mathematical content, and no new literals are introduced. In particular, the SAT solver has to compute an unsatisfiable core, and output it to the mathsolver. In contrast to MathSAT, their strategy is best understood as a dialogue between the SAT solver and the mathsolver, albeit they speak in different languages.

In an eager approach, we need to introduce new literals, since it is necessary to encode all mathematical information of the input into the SAT instance. Consequently, mathematical pre-processing is the first part of the tool chain. In the BMF tool [25], this principle is applied to linear constraints using the Fourier-Motzkin (FM) decision procedure for inequalities. Let C be a set of inequalities, and recall from Section 2 that the FM procedure generates a finite set $C^* \supseteq C$ of inequalities which have the same solution space. The computation of C^* involves elementary steps where two existing inequalities ϕ_1, ϕ_2 are composed to yield a third inequality ϕ_3 . In the pre-processing step, BMF adds the implication $l_{\phi_1} \wedge l_{\phi_2} \rightarrow l_{\phi_3}$ which can be written as a clause $\neg l_{\phi_1} \vee \neg l_{\phi_2} \vee l_{\phi_3}$. By the completeness of the FM procedure, the new (Boolean) formula is satisfiable if C is. The evident problem with this approach is the fact that C^* can become very large. Many of them are redundant, and [25] has presented techniques to prune C^* , and argued that the worst case does not appear in practice. Note that the BFM approach can be seen as a generalization of the MathSAT optimization which checks for inconsistencies between constraints in the input.

One can view the additional clauses of BFM (and also MathSAT) as “mathematical lemmas” which are added to the SAT instance before the start of the SAT solver. The usage of lemmas is also central in ICS but in a different way: when the SAT solver detects a conflict with the guidance of the mathsolver, it communicates the conflict to the mathsolver, and the mathsolver tries to compute a clause (i.e., an “inconsistency lemma”) which compactly explains the conflict, using only literals previously known to the SAT solver. Computing this clause however requires multiple calls to the mathsolver [10]. The idea of propagating entailed literals was also suggested in a forward checking technique for temporal reasoning [1].

Our approach also can be understood as lemma introduction, albeit with several crucial differences: the “lemma introduction” is done actively during the SAT propagation as integral part of the SAT solver. *We introduce lemmas not only in case of conflicts, but as a natural inference mechanism in such a way that the lemmas cover all (infinitely many) consequences, by virtue of the completeness of the Fourier-Motzkin method.* This clearly avoids the problem of underconstrained SAT instances with too many Boolean models. The “lemmas” we produce are stored in the natural way in the clause database, and enable the SAT solver to explain each conflict by a single conflict clause computation. This approach also ensures that the conflict explanation is short and mathematically concise, i.e., it exactly explains the mathematical reasoning which led to the conflict. Similarly as “classical” conflict clauses are the shortest summarizations of Boolean conflicts, our conflict clauses are the shortest summarizations of mathematical conflicts.

We conclude that there is a big spectrum of approaches for solving linear constraints which combine lazy and eager reasoning. Importantly, all the described approaches implicitly realize a specific strategy which is hardwired in the code, and reflected in the software architecture. To a large extent, this is inevitable with a decoupled architecture. The published benchmarks demonstrate that there can be no clear winner between eager and lazy approaches, i.e., the quality of the strategies depends heavily on the instances. These arguments motivate the approach described in the current paper where we describe a SAT solver which tightly integrates the math solving capabilities. Our solver can realize different strategies via the order in which decisions on literals are taken. Thus, we are able, at least in principle, to use and combine the lazy and eager strategies from the literature in a flexible manner, and to switch between them on the fly, i.e., during the solver run. It is commonly agreed that the flexibility of decision orders is crucial for the stable performance of mod-

ern SAT solvers. Consequently, we expect that the flexibility incurred by our approach deserves systematic investigation.

4 SAT with Interpreted Literals

The basic procedure and control flow of our solver is essentially identical to canonical SAT solvers, i.e., it is based on DPLL with conflict clauses. However, as soon as the SAT solver sets a *literal* x true (either by decision or by unit constraint propagation), it *interprets* the literal, i.e., it determines the *mathematical* consequences obtained from $[[x]]$ together with the previously assigned literals. We distinguish two cases:

1. If $[[x]]$ and a previously obtained $[[y]]$ together are inconsistent, i.e., if the consequence is a contradiction, we add the clause $\neg x \vee \neg y$ to the clause database.
2. If the Fourier Motzkin consequence of $[[x]]$ and $[[y]]$ is a constraint ϕ , then the new literal l_ϕ with $\text{formula}(l_\phi) = \phi$ is introduced, and the clause $\neg x \vee \neg y \vee l_\phi$, i.e., the implication $x \wedge y \rightarrow l_\phi$, is added to the clause database. If there are several independent consequences, we may add several clauses at this point.

In the first case, the new clause $\neg x \vee \neg y$ makes x and y conflicting literals, and we use the SAT solver to compute the conflict clause in exactly the same way as in canonical SAT solving, and to do backtracking accordingly. The conflict clause obtained in this way is a mathematically accurate explanation of the inconsistency between y and x . In the second case, the introduction of the clause $\neg x \vee \neg y \vee l_\phi$ has a double effect: on the one hand, the clause is already unit when it is introduced, and thus l_ϕ can be propagated; on the other hand, the clause is the justification for the SAT solver to derive ϕ , and can therefore be used for conflict clause calculation later if necessary. *Consequently, the introduction of the new literal does not increase the search space of the SAT solver, but only serves for conflict clause computation.* In fact, l_ϕ is unit by construction, and from the point of view of the SAT solver, unit propagation of l_ϕ is particularly simple, because the unit literal need not be searched for, but is already known. In case of backtracking to a level where the other literals of the clause are not assigned, the solver can safely remove the whole clause from the database. As long as only mathematically true clauses are introduced, the described procedure is sound by construction, i.e., it is able to compute some, but not necessarily all, satisfying assignments. The more complicated question of course is completeness. This depends on the underlying mathematical theory, and on the method we use to compute new clauses. In the following section, we will present a complete method for LAL which is based on Fourier-Motzkin elimination.

5 Mathematical Propagation by Fourier-Motzkin Elimination

We will now present the simpler version of our solver which does not handle equality explicitly, but treats $\mathbf{a} = \mathbf{b}$ as $\mathbf{a} \leq \mathbf{b} \wedge \mathbf{b} \leq \mathbf{a}$. Our algorithm is an instantiation of the general principle of the previous section.

Assume that the SAT solver has set a literal x true, and now wants to compute the consequences of $[[x]]$ with all previously assigned $[[y]]$'s. This is achieved by Fourier-Motzkin (FM) elimination, cf. Section 2. Recall that the basic step in FM is the elimination operation $\oplus_{\mathbf{v}}$ for a variable \mathbf{v} .

We now compute the FM consequences as follows: We loop through the arithmetical variables of $[[x]]$, and for each variable \mathbf{v} we consider all $[[y]]$'s which contain \mathbf{v} and which are not used in the derivation tree of x . For each such $[[y]]$, we test, if $[[x]] \oplus_{\mathbf{v}} [[y]]$ is applicable. As above, we distinguish two cases: First, if $[[x]] \oplus_{\mathbf{v}} [[y]]$ is a contradiction $0 < 0$, we introduce the clause $\neg x \vee \neg y$. Otherwise, we generate a new literal $l_{[[x]] \oplus_{\mathbf{v}} [[y]]}$, and the clause $\neg x \vee \neg y \vee l_{[[x]] \oplus_{\mathbf{v}} [[y]]}$. Note that in the special case where $[[x]] \oplus_{\mathbf{v}} [[y]]$ is

```

propagate(clause  $C$ , literal  $x$ )
  set  $\text{value}(x) = \text{true}$ ;
  store " $C$  implies  $x$  is true" for future conflict clause computation;
  if  $\text{type}([x]) = T_<$  then FMinterpret_literal( $x$ );
  for all clauses  $D$ , s.t.  $D$  contains  $\neg x$ 
    if  $\text{value}(D) = \text{false}$  then backtrack( $D$ );
    elseif  $D$  contains exactly one undefined "unit" literal  $u$ ;
    then propagate( $D, u$ );

```

Figure 1: The algorithm **propagate** performs the usual Boolean constraint propagation (BCP), preceded by a call to **FMinterpret_literal**.

```

FMinterpret_literal(literal  $x$ )
  for all  $v \in \text{var}([x])$ 
     $L_v := \{\text{literals } y \mid \text{value}(y) \neq \perp, v \in \text{var}([y])\}$ ;
    for all  $y \in L_v$ , s.t.  $[x] \oplus_v [y]$  is a Fourier-Motzkin step
      if  $[x] \oplus_v [y]$  is a contradiction
        then add clause  $\neg x \vee \neg y$  to database
      elseif  $\neg \exists z. \text{value}(z) \neq \perp$  and  $[z]$  subsumes  $[x] \oplus_v [y]$ 
        then create new literal  $l_{[x] \oplus_v [y]}$  with formula  $l_{[x] \oplus_v [y]} = [x] \oplus_v [y]$ ;
        add new clause  $\neg x \vee \neg y \vee l_{[x] \oplus_v [y]}$  to database;
        propagate( $\neg x \vee \neg y \vee l_{[x] \oplus_v [y]}, l_{[x] \oplus_v [y]}$ );

```

Figure 2: The algorithm **FMinterpret_literal** integrates the Fourier-Motzkin consequences into the SAT solving process.

trivially true, e.g. $0 < 5$, we need not introduce the clause. In the first case, we let the SAT solver compute a conflict clause from $\neg x \vee \neg y$, and do backtracking accordingly. In the second case, as soon as the new clause $\neg x \vee \neg y \vee l_{[x] \oplus_v [y]}$ is introduced, it becomes unit by construction as argued above, and we immediately propagate $l_{[x] \oplus_v [y]}$. Only afterwards, we move on to the next $[y]$, and later, the next v .

Consider the algorithm **propagate** in Figure 1. Except for one line of code, **propagate** is the standard BCP procedure. The crucial addition is the line which checks if the literal x indeed represents an inequality; in this case, **propagate** first calls **FMinterpret_literal**, cf. Figure 2. Essentially, **FMinterpret_literal** just adds additional clauses to the database which are inferred from the mathematical content of $[x]$. To this end, **FMinterpret_literal** computes the Fourier-Motzkin consequences of $[x]$ together with previously assigned literals $[y]$. By choosing only such literals $[y]$ from which the variable v has not been eliminated, and which contain no variables already eliminated from $[x]$, we compute only new literals which are also derivable in the standard Fourier-Motzkin elimination steps [19]. Moreover, we do not add new constraints which are weakenings (subsumptions) of previously asserted inequations, e.g. we do not add $\mathbf{a} > 0$ when we already have $\mathbf{a} > 5$.

In the full version of the paper [18] we show how the described method can be extended to handle equality propagation. Moreover, we describe a prototype implementation.

6 Conclusion

We have introduced a novel SAT-based solver for linear real number constraints. Our prototype solver is very compact, and is able to combine the advantages of the lazy and eager approaches in the literature, i.e., *by changing the decision order, we are able to perform mathematical reasoning early or late in the search tree*. We have shown how to integrate Fourier-Motzkin style reasoning directly into a SAT solver with reasonable software-engineering overhead. Our solver is thus able to compute mathematically precise explanations of

conflicts between arithmetic constraints. We are convinced that the method described here is applicable to a much wider class of theories, in particular also to uninterpreted function symbols and to integer constraints, as our approach lends itself naturally to non-convex theories.

References

- [1] A. Armando, C. Castellini, E. Giunchiglia. SAT-based procedures for temporal reasoning. In *Proc. 5th European Conf. Planning (ECP)*, 1999.
- [2] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *CADE*, pages 195–210, 2002.
- [3] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV*, pages 457–461, 2004.
- [4] C. W. Barrett. Checking validity of quantifier-free formulas in combinations of first-order theories, 2003. PhD Thesis, Stanford University.
- [5] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV*, pages 236–249, 2002.
- [6] A. Biere. Limmat solver homepage. <http://fmv.jku.at/limmat/>.
- [7] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *TACAS*, 2005. to appear.
- [8] M. Bozzano, A. Cimatti, G. Colombini, V. Kirov, and R. Sebastiani. The MathSAT solver - a progress report. In *Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2004.
- [9] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [10] L. de Moura and H. Ruess. Lemmas on demand for satisfiability solvers. In *SAT*, 2002.
- [11] L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures, 2004. available from www.icansolve.com.
- [12] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *IJCAR*, pages 218–222, 2004.
- [13] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.
- [14] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *CAV*, pages 246–249, 2001.
- [15] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 175–188, 2004.
- [16] E. Goldberg and Y. Novikov. BerkMin: A fast and robust Sat-solver. In *DATE*, pages 142–149, 2002.
- [17] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. Efficient conflict-based learning in an rtl circuit constraint solver. In *DATE*, pages 666–671, 2005.
- [18] P. Koppensteiner and H. Veith. A novel SAT procedure for linear real arithmetic. Available from www.model.in.tum.de/~veith.
- [19] H. Kuhn. Solvability and consistency for linear equations and inequalities. *American Mathematical Monthly*, 63:217–232, 1956.
- [20] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *CAV*, pages 475–478, 2004.
- [21] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.
- [22] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang. An efficient finite-domain constraint solver for circuits. In *DAC*, pages 212–217, 2004.
- [23] R. E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.
- [24] J. P. M. Silva and K. A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.
- [25] O. Strichman. On solving Presburger and linear arithmetic with SAT. In *FMCAD*, pages 160–170, 2002.
- [26] A. Stump, C. W. Barrett, and D. L. Dill. CVC: a cooperating validity checker. In *CAV*, pages 500–504, 2002.