Efficient Interpolant Generation in Satisfiability Modulo Linear Integer Arithmetic

Alberto Griggio[1] ⋆, Thi Thieu Hoa Le[2], and Roberto Sebastiani[2] ⋆⋆

[1] FBK-Irst, Trento, Italy
[2] DISI, University of Trento, Italy

Abstract. The problem of computing Craig interpolants in SAT and SMT has recently received a lot of interest, mainly for its applications in formal verification. Efficient algorithms for interpolant generation have been presented for some theories of interest —including that of equality and uninterpreted functions ($\mathcal{EUF}$), linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$), and their combination— and they are successfully used within model checking tools. For the theory of linear arithmetic over the integers ($\mathcal{LA}(\mathbb{Z})$), however, the problem of finding an interpolant is more challenging, and the task of developing efficient interpolant generators for the full theory $\mathcal{LA}(\mathbb{Z})$ is still the objective of ongoing research.
In this paper we try to close this gap. We build on previous work and present a novel interpolation algorithm for SMT($\mathcal{LA}(\mathbb{Z})$), which exploits the full power of current state-of-the-art SMT($\mathcal{LA}(\mathbb{Z})$) solvers. We demonstrate the potential of our approach with an extensive experimental evaluation of our implementation of the proposed algorithm in the MATHSAT SMT solver.

1 Motivations, related work and goals

Given two formulas A and B such that $A \wedge B$ is inconsistent, a *Craig interpolant* (simply "interpolant" hereafter) for (A, B) is a formula I s.t. A entails I, $I \wedge B$ is inconsistent, and all uninterpreted symbols of I occur in both A and B.

Interpolation in both SAT and SMT has been recognized to be a substantial tool for formal verification. For instance, in the context of software model checking based on counter-example-guided-abstraction-refinement (CEGAR) interpolants of quantifier-free formulas in suitable theories are computed for automatically refining abstractions in order to rule out spurious counterexamples. Consequently, the problem of computing interpolants in SMT has received a lot of interest in the last years (e.g., [14, 17, 19, 11, 4, 10, 13, 7, 8, 3, 12]). In the recent years, efficient algorithms and tools for interpolant generation for quantifier-free formulas in SMT have been presented for some theories of interest, including that of equality and uninterpreted functions ($\mathcal{EUF}$) [14, 7], linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$) [14, 17, 4], and for their combination [19, 17, 4, 8], and they are successfully used within model-checking tools.

⋆ Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.
⋆⋆ Supported by SRC under GRC Custom Research Project 2009-TJ-1880 WOLFLING.

For the theory of linear arithmetic over the *integers* ($\mathcal{LA}(\mathbb{Z})$), however, the problem of finding an interpolant is more challenging. In fact, it is not always possible to obtain quantifier-free interpolants starting from quantifier-free input formulas in the standard signature of $\mathcal{LA}(\mathbb{Z})$ (consisting of Boolean connectives, integer constants and the symbols $+, \cdot, \leq, =$) [14]. For instance, there is no quantifier-free interpolant for the $\mathcal{LA}(\mathbb{Z})$-formulas $A \stackrel{\text{def}}{=} (2x - y + 1 = 0)$ and $B \stackrel{\text{def}}{=} (y - 2z = 0)$.

In order to overcome this problem, different research directions have been explored. One is to restrict to important *fragments* of $\mathcal{LA}(\mathbb{Z})$ where the problem does not occur. To this extent, efficient interpolation algorithms for the Difference Logic ($\mathcal{DL}$) and Unit-Two-Variables-Per-Inequality ($\mathcal{UTVPI}$) fragments of $\mathcal{LA}(\mathbb{Z})$ have been proposed in [4]. Another direction is to extend the signature of $\mathcal{LA}(\mathbb{Z})$ to contain *modular equalities* $=_c$ (or, equivalently, *divisibility predicates*), so that it is possible to compute quantifier-free $\mathcal{LA}(\mathbb{Z})$ interpolants by means of quantifier elimination —which is however prohibitively expensive in general, both in theory and in practice. For instance, $I \stackrel{\text{def}}{=} (-y + 1 =_2 0) \equiv \exists x.(2x - y + 1 = 0)$ is an interpolant for the formulas (A, B) above. Using modular equalities, Jain et al. [10] developed polynomial-time interpolation algorithms for linear equations and their negation and for linear modular equations. A similar algorithm was also proposed in [13]. The work in [3] was the first to present an interpolation algorithm for the full $\mathcal{LA}(\mathbb{Z})$ (augmented with divisibility predicates) which was not based on quantifier elimination. Finally, an alternative algorithm, exploiting efficient interpolation procedures for $\mathcal{LA}(\mathbb{Q})$ and for linear equations in $\mathcal{LA}(\mathbb{Z})$, has been recently presented in [12].

The obvious limitation of the first research direction is that it does not cover the full $\mathcal{LA}(\mathbb{Z})$. For the second direction, the approaches so far seem to suffer from some drawbacks. In particular, some of the interpolation rules of [3] might result in an exponential blow-up in the size of the interpolants wrt. the size of the proofs of unsatisfiability from which they are generated. The algorithm of [12] avoids this, but at the cost of significantly restricting the heuristics commonly used in state-of-the-art SMT solvers for $\mathcal{LA}(\mathbb{Z})$ (e.g. in the framework of [12] both the use of Gomory cuts [18] and of "cuts from proofs" [5] is not allowed). More in general, the important issue of how to efficiently integrate the presented techniques into a state-of-the-art SMT($\mathcal{LA}(\mathbb{Z})$) solver is not immediate to foresee from the papers.

In this paper we try to close this gap. After recalling the necessary background knowledge (§2), we present our contribution, which is twofold.

First (§3) we show how to extend the state-of-the art $\mathcal{LA}(\mathbb{Z})$-solver of MATHSAT [9] in order to implement interpolant generation on top of it without affecting its efficiency. To this extent, we combine different algorithms corresponding to the different submodules of the $\mathcal{LA}(\mathbb{Z})$-solver, so that each of the submodules requires only minor modifications, and implement them in MATHSAT (MATHSAT-MODEQ hereafter). An extensive empirical evaluation (§5) shows that MATHSAT-MODEQ outperforms in efficiency all existing interpolant generators for $\mathcal{LA}(\mathbb{Z})$.

Second (§4), we propose a novel and general interpolation algorithm for $\mathcal{LA}(\mathbb{Z})$, independent from the architecture of MATHSAT, which overcomes the drawbacks of the current approaches. The key idea is to extend both the signature and the domain of $\mathcal{LA}(\mathbb{Z})$: we extend the signature by adding the *ceiling function* $\lceil\cdot\rceil$ to it, and the domain

by allowing non-variable terms to be non-integers. This greatly simplifies the interpolation procedure, and allows for producing interpolants which are much more compact than those generated by the algorithm of [3]. Also this novel technique was easily implemented on top of the $\mathcal{LA}(\mathbb{Z})$-solver of MATHSAT without affecting its efficiency. (We call this implementation MATHSAT-CEIL.) An extensive empirical evaluation (§5) shows that MATHSAT-CEIL drastically outperforms MATHSAT-MODEQ, and hence all other existing interpolant generators for $\mathcal{LA}(\mathbb{Z})$, for both efficiency and size of the final interpolant.

2 Background: SMT($\mathcal{LA}(\mathbb{Z})$)

2.1 Generalities

Satisfiability Modulo Theory – SMT. Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We call *Satisfiability Modulo (the) Theory* $\mathcal{T}$, SMT($\mathcal{T}$), the problem of deciding the satisfiability of quantifier-free formulas wrt. a background theory $\mathcal{T}$. [3] Given a theory $\mathcal{T}$, we write $\phi \models_{\mathcal{T}} \psi$ (or simply $\phi \models \psi$) to denote that the formula ψ is a logical consequence of ϕ in the theory $\mathcal{T}$. With $\phi \preceq \psi$ we denote that all uninterpreted (in $\mathcal{T}$) symbols of ϕ appear in ψ. With a little abuse of notation, we might sometimes denote conjunctions of literals $l_1 \wedge \ldots \wedge l_n$ as sets $\{l_1, \ldots, l_n\}$ and vice versa. If η is the set $\{l_1, \ldots, l_n\}$, we might write $\neg\eta$ to mean $\neg l_1 \vee \ldots \vee \neg l_n$.

We call $\mathcal{T}$-solver a procedure that decides the consistency of a conjunction of literals in $\mathcal{T}$. If S is a set of literals in $\mathcal{T}$, we call *$\mathcal{T}$-conflict set w.r.t. S* any subset η of S which is inconsistent in $\mathcal{T}$. We call $\neg\eta$ a *$\mathcal{T}$-lemma* (notice that $\neg\eta$ is a $\mathcal{T}$-valid clause).

A standard technique for solving the SMT($\mathcal{T}$) problem is to integrate a DPLL-based SAT solver and a $\mathcal{T}$-solver in a *lazy* manner (see, e.g., [2] for a detailed description). DPLL is used as an enumerator of truth assignments for the propositional abstraction of the input formula. At each step, the set of $\mathcal{T}$-literals in the current assignment is sent to the $\mathcal{T}$-solver to be checked for consistency in $\mathcal{T}$. If S is inconsistent, the $\mathcal{T}$-solver returns a conflict set η, and the corresponding $\mathcal{T}$-lemma $\neg\eta$ is added as a blocking clause in DPLL, and used to drive the backjumping and learning mechanism.

Interpolation in SMT. We consider the SMT($\mathcal{T}$) problem for some background theory $\mathcal{T}$. Given an ordered pair (A, B) of formulas such that $A \wedge B \models_{\mathcal{T}} \perp$, a *Craig interpolant* (simply "interpolant" hereafter) is a formula I s.t. (i) $A \models_{\mathcal{T}} I$, (ii) $I \wedge B$ is $\mathcal{T}$-inconsistent, and (iii) $I \preceq A$ and $I \preceq B$.

Following [14], an interpolant for (A, B) in SMT($\mathcal{T}$) can be generated by combining a propositional interpolation algorithm for the Boolean structure of the formula $A \wedge B$ with a $\mathcal{T}$-specific interpolation procedure that deals only with negations of $\mathcal{T}$-lemmas, that is, with $\mathcal{T}$-inconsistent conjunctions of $\mathcal{T}$-literals. Therefore, in the rest of the paper, we shall consider algorithms for conjunctions/sets of literals only, which can be extended to general formulas by simply "plugging" them into the algorithm of [14].

[3] The general definition of SMT deals also with quantified formulas. Nevertheless, in this paper we restrict our interest to quantifier-free formulas.

2.2 Efficient SMT($\mathcal{LA}(\mathbb{Z})$) solving

In this section, we describe our algorithm for efficiently solving SMT($\mathcal{LA}(\mathbb{Z})$) problems, as implemented in the MATHSAT 5 SMT solver [9]. 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 to overcome the limitations of each of them. Both the experimental results of [9] and the latest SMT solvers competition SMT-COMP'10 [4] demonstrate that it represents the current state of the art in SMT($\mathcal{LA}(\mathbb{Z})$).

The architecture of the solver is outlined in Fig. 1. It is organized as a layered hierarchy of submodules, with cheaper (but less powerful) ones invoked earlier and more often. The general strategy used for checking the consistency of a set of $\mathcal{LA}(\mathbb{Z})$-constraints is as follows.

First, the rational relaxation of the problem is checked, using a Simplex-based $\mathcal{LA}(\mathbb{Q})$-solver similar to that described in [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 $\mathcal{LA}(\mathbb{Z})$ equations (Diophantine equations) is invoked. This module is similar to the first part of the Omega test described in [16]: 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 itself is infeasible, this module is also able to detect the inconsistency.

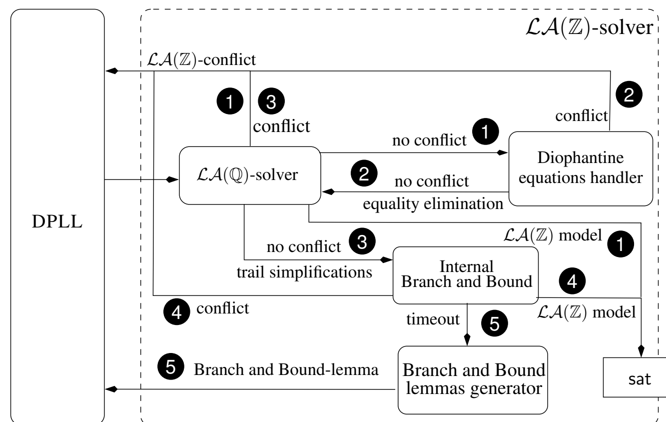

Fig. 1. Architecture of the $\mathcal{LA}(\mathbb{Z})$-solver of MATHSAT.

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 [18]. 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

[4] `http://www.smtcomp.org/2010/`

which the "external" branch and bound module is called. This works in cooperation with the DPLL engine, using the "splitting on-demand" approach of [1]: case splits are delegated to DPLL, by sending to it $\mathcal{LA}(\mathbb{Z})$-valid clauses of the form $(t - c \leq 0) \vee (-t + c + 1 \leq 0)$ (called branch-and-bound lemmas) that encode the required splits. Such clauses are generated with the "cuts from proofs" algorithm of [5]: "normal" branch-and-bound steps – splitting cases on an individual variable – are interleaved with "extended" steps, in which branch-and-bound lemmas involve an arbitrary linear combination of variables, generated by computing proofs of unsatisfiability of particular systems of Diophantine equations.

3 From $\mathcal{LA}(\mathbb{Z})$-solving to $\mathcal{LA}(\mathbb{Z})$-interpolation

Our objective is that of devising an interpolation algorithm that could be implemented on top of the $\mathcal{LA}(\mathbb{Z})$-solver described in the previous section without affecting its efficiency. To this end, we combine different algorithms corresponding to the different submodules of the $\mathcal{LA}(\mathbb{Z})$-solver, so that each of the submodules requires only minor modifications.

3.1 Interpolation for Diophantine equations

An interpolation procedure for systems of Diophantine equations was given by Jain et al. in [10]. The procedure starts from a proof of unsatisfiability expressed as a linear combination of the input equations whose result is an equation $(\sum c_i x_i + c = 0)$ in which the greatest common divisor (GCD) of the coefficients c_i of the variables is not a divisor of the constant term c.

Given a proof of unsatisfiability for a system of equations partitioned into A and B, let $(\sum_{x_i \in A \cap B} c_i x_i + \sum_{y_j \notin B} a_j y_j + c = 0)$ be the linear combination of the equations from A with the coefficients given by the proof of unsatisfiability. Then, $I \stackrel{\text{def}}{=} \sum_{x_i \in A \cap B} c_i x_i + c =_g 0$, where g is any integer that divides $GCD(a_j)$, is an interpolant for (A, B) [10]. Jain et al. show that a proof of unsatisfiability can be obtained by computing the Hermite Normal Form [18] of the system of equations. However, this is only one possible way of obtaining such proof. In particular, as shown in [9], the submodule of our $\mathcal{LA}(\mathbb{Z})$-solver that deals with Diophantine equations can already produce proofs of unsatisfiability directly. Therefore, we can apply the interpolation algorithm of [10] without any modification to the solver.

3.2 Interpolation for inequalities

The second submodule of our $\mathcal{LA}(\mathbb{Z})$-solver checks the $\mathcal{LA}(\mathbb{Q})$-consistency of a set of inequalities, some of which obtained by *substitution* and *tightening* [9]. In this case, we produce interpolants starting from proofs of unsatisfiability in the *cutting-plane proof system*, a complete proof system for $\mathcal{LA}(\mathbb{Z})$ [18]. Similarly to previous work on $\mathcal{LA}(\mathbb{Q})$ and $\mathcal{LA}(\mathbb{Z})$ [14, 3], we produce interpolants by annotating each step of the proof of unsatisfiability of $A \wedge B$, such that the annotation for the root of the proof (deriving an inconsistent inequality $(c \leq 0)$ with $c \in \mathbb{Z}^{>0}$) is an interpolant for (A, B).

Definition 1 (Valid annotated sequent). *An* annotated sequent *is a sequent in the form* $(A, B) \vdash (t \leq 0)[I]$ *where* A *and* B *are conjunctions of equalities and inequalities in* $\mathcal{LA}(\mathbb{Z})$*, and where* I *(called* annotation*) is a set of pairs* $\langle (t_i \leq 0), E_i \rangle$ *in which* E_i *is a (possibly empty) conjunction of equalities and modular equalities. It is said to be* valid *when:*

1. $A \models \bigvee_{\langle t_i \leq 0, E_i \rangle \in I} ((t_i \leq 0) \wedge E_i)$*;*
2. *For all* $\langle t_i \leq 0, E_i \rangle \in I$*,* $B \wedge E_i \models (t - t_i \leq 0)$*;*
3. *For every element* $\langle (t_i \leq 0), E_i \rangle$ *of* I*,* $t_i \preceq A$*,* $(t - t_i) \preceq B$*,* $E_i \preceq A$ *and* $E_i \preceq B$*.*

Definition 2 (Interpolating Rules). *The* $\mathcal{LA}(\mathbb{Z})$*-interpolating inference rules that we use are the following:*

Hyp-A $\dfrac{}{(A, B) \vdash (t \leq 0)[\{\langle t \leq 0, \top \rangle\}]}$ *if* $(t \leq 0) \in A$ *or* $(t = 0) \in A$

Hyp-B $\dfrac{}{(A, B) \vdash (t \leq 0)[\{\langle 0 \leq 0, \top \rangle\}]}$ *if* $(t \leq 0) \in B$ *or* $(t = 0) \in B$

Comb $\dfrac{(A, B) \vdash t_1 \leq 0[I_1] \quad (A, B) \vdash t_2 \leq 0[I_2]}{(A, B) \vdash (c_1 t_1 + c_2 t_2 \leq 0)[I]}$ *where:*
- $c_1, c_2 > 0$
- $I \stackrel{def}{=} \{\langle c_1 t'_1 + c_2 t'_2 \leq 0, E_1 \wedge E_2 \rangle \mid \langle t'_1 \leq 0, E_1 \rangle \in I_1$ *and* $\langle t'_2 \leq 0, E_2 \rangle \in I_2\}$

Strengthen $\dfrac{(A, B) \vdash \sum_i c_i x_i + c \leq 0[\{\langle t' \leq 0, \top \rangle\}]}{(A, B) \vdash \sum_i c_i x_i + c + k \leq 0[I]}$ *where:*
- $k \stackrel{def}{=} d \left\lceil \frac{c}{d} \right\rceil - c$*, and* $d > 0$ *is an integer that divides all the* c_i*'s;*
- $I \stackrel{def}{=} \{\langle t' + j \leq 0, \exists (x \not\in B).(t' + j = 0) \rangle \mid 0 \leq j < k\} \cup \{\langle t' + k \leq 0, \top \rangle\}$*; and*
- $\exists (x \not\in B).(t' + j = 0)$ *denotes the result of the existential elimination from* $(t' + j = 0)$ *of all and only the variables* $x_1, ..., x_n$ *not occurring in* B*.*[5]

Theorem 1. *All the interpolating rules preserve the validity of the sequents.*

Corollary 1. *If we can derive a valid sequent* $(A, B) \vdash c \leq 0[I]$ *with* $c \in \mathbb{Z}^{>0}$*, then* $\varphi_I \stackrel{def}{=} \bigvee_{\langle t_i \leq 0, E_i \rangle \in I} ((t_i \leq 0) \wedge E_i)$ *is an interpolant for* (A, B)*.*

Notice that the first three rules correspond to the rules for $\mathcal{LA}(\mathbb{Q})$ given in [14], whereas Strengthen is a reformulation of the k-Strengthen rule given in [3]. Moreover, although the rules without annotations are refutationally complete for $\mathcal{LA}(\mathbb{Z})$, in the above formulation the annotation of Strengthen might prevent its applicability, thus losing completeness. In particular, it only allows to produce proofs with at most one strengthening per branch. Such restriction has been put only for simplifying the proofs of correctness, and it is not present in the original k-Strengthen of [3]. However, for our purposes this is not a problem, since we use the above rules only in the second submodule of our $\mathcal{LA}(\mathbb{Z})$-solver, which always produces proofs with at most one strengthening per branch.

[5] We recall that $\exists (x_1, \ldots, x_n).(\sum_i c_i x_i + \sum_j d_j y_j + c = 0) \equiv (\sum_j d_j y_j + c =_{GCD(c_i)} 0)$, and that $(t =_0 0) \equiv (t = 0)$.

Generating cutting-plane proofs in the $\mathcal{LA}(\mathbb{Z})$-solver. The equality elimination and tightening step generates new inequalities $(t' + c' + k \leq 0)$ starting from a set of input equalities $\{e_1 = 0, \ldots, e_n = 0\}$ and an input inequality $(t+c \leq 0)$. Thanks to its proof-production capabilities [9], we can extract from the Diophantine equations submodule the coefficients $\{c_1, \ldots, c_n\}$ such that $(\sum_i c_i e_i + t + c \leq 0) \equiv (t' + c' \leq 0)$. Thus, we can generate a proof of $(t' + c' \leq 0)$ by using the Comb and Hyp rules. We then use the Strengthen rule to obtain a proof of $(t' + c' + k \leq 0)$. The new inequalities generated are then added to the $\mathcal{LA}(\mathbb{Q})$-solver. If a $\mathcal{LA}(\mathbb{Q})$-conflict is found, then, the $\mathcal{LA}(\mathbb{Q})$-solver produces a $\mathcal{LA}(\mathbb{Q})$-proof of unsatisfiability (as described in [4]) in which some of the leaves are the new inequalities generated by equality elimination and tightening. We can then simply replace such leaves with the corresponding cutting-plane proofs to obtain the desired cutting-plane unsatisfiability proof.

3.3 Interpolation with branch-and-bound

Interpolation via splitting on-demand. In the splitting on-demand approach, the $\mathcal{LA}(\mathbb{Z})$ solver might not always detect the unsatisfiability of a set of constraints by itself; rather, it might cooperate with the DPLL solver by asking it to perform some case splits, by sending to DPLL some additional $\mathcal{LA}(\mathbb{Z})$-lemmas encoding the different case splits. In our interpolation procedure, we must take this possibility into account.

Let $(t - c \leq 0) \vee (-t + c + 1 \leq 0)$ be a branch-and-bound lemma added to the DPLL solver by the $\mathcal{LA}(\mathbb{Z})$-solver, using splitting on-demand. If $t \preceq A$ or $t \preceq B$, then we can exploit the Boolean interpolation algorithm also for computing interpolants in the presence of splitting-on-demand lemmas. The key observation is that the lemma $(t - c \leq 0) \vee (-t + c + 1 \leq 0)$ is a *valid clause* in $\mathcal{LA}(\mathbb{Z})$. Therefore, we can add it to any formula without affecting its satisfiability. Thus, if $t \preceq A$ we can treat the lemma as a clause from A, and if $t \preceq B$ we can treat it as a clause from B.[6]

Thanks to the observation above, in order to be able to produce interpolants with splitting on-demand the only thing we need is to make sure that we do not generate lemmas containing AB-mixed terms.[7] This is always the case for "normal" branch-and-bound lemmas (since they involve only one variable), but this is not true in general for "extended" branch-and-bound lemmas generated from proofs of unsatisfiability using the "cuts from proofs" algorithm of [5]. The following example shows one such case.

Example 1. Let A and B be defined as

$$A \stackrel{\text{def}}{=} (x - 2y \leq 0) \wedge (2y - x \leq 0), \quad B \stackrel{\text{def}}{=} (x - 2z - 1 \leq 0) \wedge (2z + 1 - x \leq 0)$$

When solving $A \wedge B$ using extended branch and bound, we might generate the following AB-mixed lemma: $(y - z \leq 0) \vee (-y + z + 1 \leq 0)$.

Since we want to be able to reuse the Boolean interpolation algorithm also for splitting on-demand, we want to avoid generating AB-mixed lemmas. However, we would still like to exploit the cuts from proofs algorithm of [5] as much as possible. We describe how we do this in the following.

---

[6] If both $t \preceq A$ and $t \preceq B$, we are free to choose between the two alternatives.

[7] That is, terms t such that $t \not\preceq A$ and $t \not\preceq B$.

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

$$(\textstyle\sum_i \frac{c'_i}{g} x_i \leq \left\lceil \frac{-c'_i}{g} \right\rceil - 1) \vee (\left\lceil \frac{-c'_i}{g} \right\rceil \leq \textstyle\sum_i \frac{c'_i}{g} x_i).$$

Avoiding AB-mixed lemmas. If $\sum_i \frac{c'_i}{g} x_i$ is not AB-mixed, we can generate the above lemma also when computing interpolants. If $\sum_i \frac{c'_i}{g} x_i$ is AB-mixed, instead, we generate a different lemma, still exploiting the unsatisfiability of (the equations corresponding to) the defining constraints. Since D_E is unsatisfiable, we know that the current rational solution μ is not compatible with the current set of defining constraints. If the defining constraints were all equations, the submodule for handling Diophantine equations would have detected the conflict. Therefore, there is at least one defining constraint $\sum_i \bar{c}_i x_i + \bar{c} \leq 0$. Our idea is that of *splitting* this constraint into $(\sum_i \bar{c}_i x_i + \bar{c} + 1 \leq 0)$ and $(\sum_i \bar{c}_i x_i + \bar{c} = 0)$, by generating the lemma

$$\neg(\textstyle\sum_i \bar{c}_i x_i + \bar{c} \leq 0) \vee (\textstyle\sum_i \bar{c}_i x_i + \bar{c} + 1 \leq 0) \vee (\textstyle\sum_i \bar{c}_i x_i + \bar{c} = 0).$$

In this way, we are either "moving away" from the current bad rational solution μ (when $(\sum_i \bar{c}_i x_i + \bar{c} + 1 \leq 0)$ is set to true), or we are forcing one more element of the set of defining constraints to be an equation (when $(\sum_i \bar{c}_i x_i + \bar{c} = 0)$ is set to true): if we repeat the splitting, then, eventually all the defining constraints for the bad solution μ will be equations, thus allowing the Diophantine equations handler to detect the conflict without the need of generating more branch-and-bound lemmas. Since the set of defining constraints is a subset of the input constraints, lemmas generated in this way will never be AB-mixed.

It should be mentioned that this procedure is very similar to the algorithm used in the recent work [12] for avoiding the generation of AB-mixed cuts. However, the criterion used to select which inequality to split and how to split it is different (in [12] such inequality is selected among those that are violated by the closest integer solution to the current rational solution). Moreover, we don't do this systematically, but rather only if the cuts from proofs algorithm is not able to generate a non-AB-mixed lemma by itself. In a sense, the approach of [12] is "pessimistic" in that it systematically excludes certain kinds of cuts, whereas our approach is more "optimistic".

Interpolation for the internal branch-and-bound module. From the point of view of interpolation the subdivision of the branch-and-bound module in an "internal" and an "external" part poses no difficulty. The only difference between the two is that in the former the case splits are performed by the $\mathcal{LA}(\mathbb{Z})$-solver instead of DPLL. However, we can still treat such case splits as if they were performed by DPLL, build a Boolean resolution proof for the $\mathcal{LA}(\mathbb{Z})$-conflicts discovered by the internal branch-and-bound procedure, and then apply the propositional interpolation algorithm as in the case of splitting on-demand.

4 A novel general interpolation technique for inequalities

The use of the Strenghen rule allows us to produce interpolants with very little modifications to the $\mathcal{LA}(\mathbb{Z})$-solver (we only need to enable the generation of cutting-plane proofs), which in turn result in very little overhead *at search time*. However, the Strengthen rule might cause a very significant overhead *when generating the interpolant* from a proof of unsatisfiability. In fact, even a single Strengthen application results in a disjunction whose size is proportional to the *value of the constant* k in the rule. The following example, taken from [12], illustrates the problem.

Example 2. Consider the following (parametric) interpolation problem [12]:

$$A \stackrel{\text{def}}{=} (-y - 2nx - n + 1 \leq 0) \wedge (y + 2nx \leq 0)$$
$$B \stackrel{\text{def}}{=} (-y - 2nz + 1 \leq 0) \wedge (y + 2nz - n \leq 0)$$

where the parameter n is an integer constant greater than 1. Using the rules of §3.2, we can construct the following annotated cutting-plane proof of unsatisfiability:

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc} y + 2nx \leq 0 & -y - 2nz + 1 \leq 0 \\ [\{\langle y + 2nx \leq 0, \top\rangle\}] & [\{\langle 0 \leq 0, \top\rangle\}] \end{array}}{\begin{array}{c} 2nx - 2nz + 1 \leq 0 \\ [\{\langle y + 2nx \leq 0, \top\rangle\}] \end{array}}}{\begin{array}{l} 2nx - 2nz + 1 + (2n-1) \leq 0 \\ [\{\langle y + 2nx + j \leq 0, \\ \quad \exists x.(y + 2nx + j = 0)\rangle \mid \\ \quad 0 \leq j < 2n-1\} \cup \\ \ \{\langle y + 2nx + 2n - 1 \leq 0, \top\rangle\}] \end{array}} \qquad \dfrac{\begin{array}{cc} -y - 2nx - n + 1 \leq 0 & y + 2nz - n \leq 0 \\ [\{\langle -y - 2nx - n + 1 \leq 0, \top\rangle\}] & [\{\langle 0 \leq 0, \top\rangle\}] \end{array}}{\begin{array}{c} -2nx + 2nz - 2n + 1 \leq 0 \\ [\{\langle -y - 2nx - n + 1 \leq 0, \top\rangle\}] \end{array}}}{1 \leq 0 \ \begin{array}{l} [\{\langle j - n + 1 \leq 0, \exists x.(y + 2nx + j = 0)\rangle \mid 0 \leq j < 2n-1\} \cup \\ \ \{\langle (2n-1) - n + 1 \leq 0, \top\rangle\}] \end{array}}$$

By observing that $(j - n + 1 \leq 0) \models \bot$ when $j \geq n$, the generated interpolant is:

$$(y =_{2n} -n + 1) \vee (y =_{2n} -n + 2) \vee \ldots \vee (y =_{2n} 0),$$

whose size is linear in n, and thus exponential wrt. the size of the input problem. In fact, in [12], it is said that this is the only (up to equivalence) interpolant for (A, B) that can be obtained by using only interpreted symbols in the signature $\Sigma \stackrel{\text{def}}{=} \{=, \leq, +, \cdot\} \cup \mathbb{Z} \cup \{=_g \mid g \in \mathbb{Z}^{>0}\}$.

In order to overcome this drawback, we present a novel and very effective way of computing interpolants in $\mathcal{LA}(\mathbb{Z})$, which is inspired by a result by Pudlák [15]. The key idea is *to extend both the signature and the domain* of the theory by explicitly introducing the *ceiling function* $\lceil\cdot\rceil$ and by allowing non-variable terms to be non-integers.

As in the previous Section, we use the annotated rules Hyp-A, Hyp-B and Comb. However, in this case the annotations are *single* inequalities in the form $(t \leq 0)$ rather than (possibly large) sets of inequalities and equalities. Moreover, we replace the Strenghten rule with the equivalent Division rule:

Division

$$\frac{(A,B) \vdash \sum_i a_i x_i + \sum_j c_j y_j + \sum_k b_k z_k + c \leq 0 \; [\sum_i a_i x_i + \sum_j c'_j y_j + c' \leq 0]}{(A,B) \vdash \sum_i \frac{a_i}{d} x_i + \sum_j \frac{c_j}{d} y_j + \sum_k \frac{b_k}{d} z_k + \left\lceil \frac{c}{d} \right\rceil \leq 0 \; [\sum_i \frac{a_i}{d} x_i + \left\lceil \frac{\sum_j c'_j y_j + c'}{d} \right\rceil \leq 0]}$$

where:
- $x_i \notin B$, $y_j \in A \cap B$, $z_k \notin A$
- $d > 0$ divides all the a_i's, c_j's and b_k's

As before, if we ignore the presence of annotations, the rules Hyp-A, Hyp-B, Comb and Division form a complete proof systems for $\mathcal{LA}(\mathbb{Z})$ [18]. Notice also that all the rules Hyp-A, Hyp-B, Comb and Division preserve the following invariant: the coefficients a_i of the A-local variables are always the same for the implied inequality and its annotation. This makes the Division rule always applicable. Therefore, the above rules can be used to annotate any cutting-plane proof. In particular, this means that our new technique can be applied also to proofs generated by other $\mathcal{LA}(\mathbb{Z})$ techniques used in modern SMT solvers, such as those based on Gomory cuts or on the Omega test [16].

Definition 3. *An annotated sequent* $(A, B) \vdash (t \leq 0)[(t' \leq 0)]$ *is* valid *when:*

1. $A \models (t' \leq 0)$*;*
2. $B \models (t - t' \leq 0)$*;*
3. $t' \preceq A$ *and* $(t - t') \preceq B$*.*

Theorem 2. *All the interpolating rules preserve the validity of the sequents.*

Corollary 2. *If we can derive a valid sequent* $(A, B) \vdash c \leq 0[t \leq 0]$ *with* $c > 0$*, then* $(t \leq 0)$ *is an interpolant for* (A, B)*.*

Example 3. Consider the following interpolation problem:

$$A \stackrel{\text{def}}{=} (y = 2x), \qquad B \stackrel{\text{def}}{=} (y = 2z + 1).$$

The following is an annotated cutting-plane proof of unsatisfiability for $A \wedge B$:

$$\dfrac{\dfrac{\dfrac{\dfrac{y = 2x}{y - 2x \leq 0[y - 2x \leq 0]} \quad \dfrac{y = 2z + 1}{2z + 1 - y \leq 0[0 \leq 0]}}{2z - 2x + 1 \leq 0[y - 2x \leq 0]}}{z - x + 1 \leq 0[-x + \lceil \frac{y}{2} \rceil \leq 0]} \quad \dfrac{\dfrac{y = 2x}{\begin{array}{c}2x - y \leq 0\\ {[2x - y \leq 0]}\end{array}} \quad \dfrac{y = 2z + 1}{\begin{array}{c}y - 2z - 1 \leq 0\\ {[0 \leq 0]}\end{array}}}{2x - 2z - 1 \leq 0[2x - y \leq 0]}}{1 \leq 0[-y + 2 \lceil \frac{y}{2} \rceil \leq 0]}$$

Then, $(-y + 2\left\lceil \frac{y}{2} \right\rceil \leq 0)$ is an interpolant for (A, B).

Using the ceiling function, we do not incur in any blowup of the size of the generated interpolant wrt. the size of the proof of unsatisfiability.[8] In particular, by using the ceiling function we might produce interpolants which are up to exponentially smaller than those generated using modular equations. The intuition is that the use of the ceiling function in the annotation of the Division rule allows for expressing *symbolically* the case distinction that the Strengthen rule of §3.2 was expressing *explicitly* as a disjunction of modular equations.

Example 4. Consider again the parametric interpolation problem of Example 2:

$$A \stackrel{\text{def}}{=} (-y - 2nx - n + 1 \leq 0) \wedge (y + 2nx \leq 0)$$
$$B \stackrel{\text{def}}{=} (-y - 2nz + 1 \leq 0) \wedge (y + 2nz - n \leq 0)$$

Using the ceiling function, we can generate the following annotated proof:

$$\dfrac{\dfrac{\dfrac{\begin{array}{cc} y + 2nx \leq 0 & -y - 2nz + 1 \leq 0 \\ {[y + 2nx \leq 0]} & {[0 \leq 0]} \end{array}}{\begin{array}{c} 2nx - 2nz + 1 \leq 0 \\ {[y + 2nx \leq 0]} \end{array}}}{\begin{array}{c} 2n \cdot (x - z + 1 \leq 0) \\ {[x + \lceil \frac{y}{2n} \rceil \leq 0]} \end{array}} \qquad \dfrac{\begin{array}{cc} -y - 2nx - n + 1 \leq 0 & y + 2nz - n \leq 0 \\ {[-y - 2nx - n + 1 \leq 0]} & {[0 \leq 0]} \end{array}}{\begin{array}{c} -2nx + 2nz - 2n + 1 \leq 0 \\ {[-y - 2nx - n + 1 \leq 0]} \end{array}}}{1 \leq 0 \; [2n \lceil \frac{y}{2n} \rceil - y - n + 1 \leq 0]}$$

The interpolant corresponding to such proof is then $(2n \left\lceil \frac{y}{2n} \right\rceil - y - n + 1 \leq 0)$, whose size is linear in the size of the input.

Solving and interpolating formulas with ceilings. Any SMT solver supporting $\mathcal{LA}(\mathbb{Z})$ can be easily extended to support formulas containing ceilings. In fact, we notice that we can eliminate ceiling functions from a formula φ with a simple preprocessing step as follows:

1. Replace every term $\lceil t_i \rceil$ occurring in φ with a fresh integer variable $x_{\lceil t_i \rceil}$;
2. Set φ to $\varphi \wedge \bigwedge_i \{(x_{\lceil t_i \rceil} - 1 < t_i \leq x_{\lceil t_i \rceil})\}$.

Moreover, we remark that for using ceilings we must only be able to *represent* non-variable terms with rational coefficients, but we don't need to extend our $\mathcal{LA}(\mathbb{Z})$-solver to support Mixed Rational/Integer Linear Arithmetic. This is because, after the elimination of ceilings performed during preprocessing, we can multiply both sides of the introduced constraints $(x_{\lceil t_i \rceil} - 1 < t_i)$ and $(t_i \leq x_{\lceil t_i \rceil})$ by the least common multiple of the rational coefficients in t_i, thus obtaining two $\mathcal{LA}(\mathbb{Z})$-inequalities.

For interpolation, it is enough to preprocess A and B separately, so that the elimination of ceilings will not introduce variables common to A and B.

[8] However, we remark that, in general, cutting-plane proofs of unsatisfiability can be exponentially large wrt. the size of the input problem [18, 15].

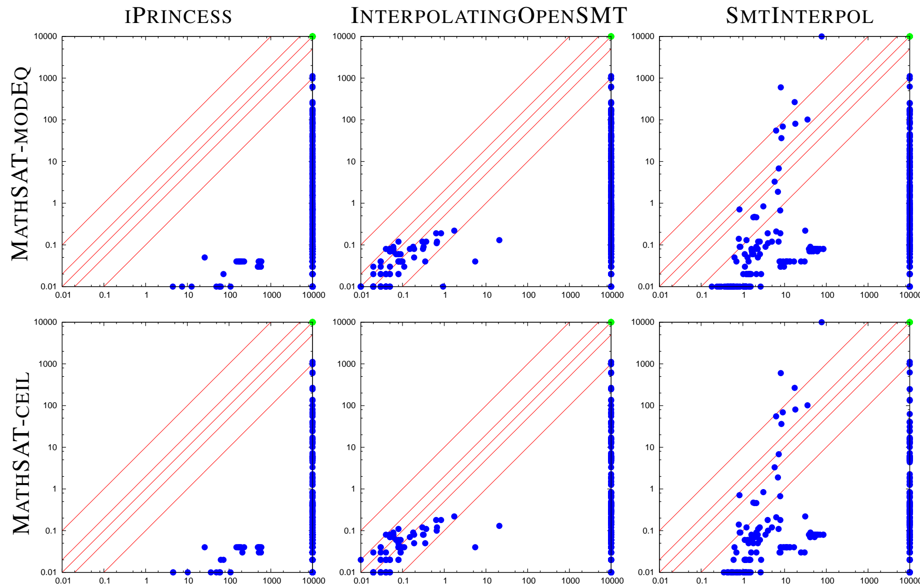

Fig. 2. Comparison between MathSAT and the other $\mathcal{LA}(\mathbb{Z})$-interpolating tools, execution time.

5 Experimental evaluation

The techniques presented in previous sections have been implemented within the MathSAT 5 SMT solver [9]. In this section, we experimentally evaluate our approach.

5.1 Description of the benchmark sets

We have performed our experiments on a subset of the benchmark instances in the QF_LIA ("quantifier-free $\mathcal{LA}(\mathbb{Z})$") category of the SMT-LIB.[9] More specifically, we have selected the subset of $\mathcal{LA}(\mathbb{Z})$-unsatisfiable instances whose rational relaxation is (easily) satisfiable, so that $\mathcal{LA}(\mathbb{Z})$-specific interpolation techniques are put under stress. In order to generate interpolation problems, we have split each of the collected instances in two parts A and B, by collecting about 40% of the toplevel conjuncts of the instance to form A, and making sure that A contains some symbols not occurring in B (so that A is never a "trivial" interpolant). In total, our benchmark set consists of 513 instances.

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 1200 seconds and a memory limit of 3 GB.

All the benchmark instances and the executable of MathSAT used to perform the experiments are available at `http://es.fbk.eu/people/griggio/papers/tacas11_experiments.tar.bz2`

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

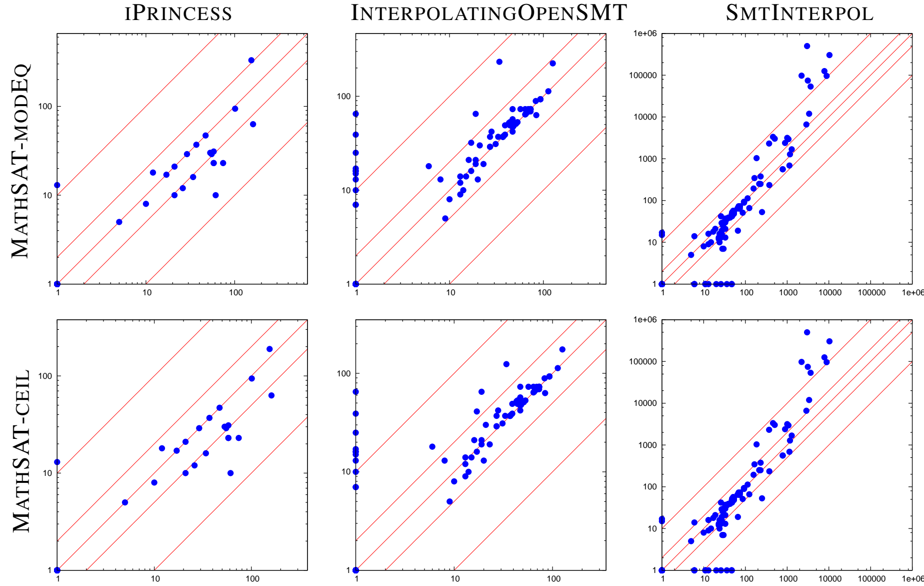

Fig. 3. Comparison between MATHSAT and the other $\mathcal{LA}(\mathbb{Z})$-interpolating tools, interpolants size (measured in number of nodes in the DAG of the interpolant). (See also footnote 13.)

5.2 Comparison with the state-of-the-art tools available

We compare MATHSAT with all the other interpolant generators for $\mathcal{LA}(\mathbb{Z})$ which are available (to the best of our knowledge): IPRINCESS [3],[10] INTERPOLATINGOPENSMT [12],[11] and SMTINTERPOL[12]. We compare not only the execution times for generating interpolants, but also the size of the generated formulas (measured in terms of number of nodes in their DAG representation).

For MATHSAT, we use two configurations: MATHSAT-MODEQ, which produces interpolants with modular equations using the Strengthen rule of §3, and MATHSAT-CEIL, which uses the ceiling function and the Division rule of §4.

Results on execution times for generating interpolants are reported in Fig. 2. Both MATHSAT-MODEQ and MATHSAT-CEIL could successfully generate an interpolant for 478 of the 513 interpolation problems (timing out on the others), whereas IPRINCESS, INTERPOLATINGOPENSMT and SMTINTERPOL were able to successfully produce an interpolant in 62, 192 and 217 cases respectively. Therefore, MATHSAT can solve more than twice as many instances as its closer competitor SMTINTERPOL, and in most cases with a significantly shorter execution time (Fig. 2).

For the subset of instances which could be solved by at least one other tool, therefore, the two configurations of MATHSAT seem to perform equally well. The situation

[10] `http://www.philipp.ruemmer.org/iprincess.shtml`

[11] `http://www.philipp.ruemmer.org/interpolating-opensmt.shtml`

[12] `http://swt.informatik.uni-freiburg.de/research/tools/smtinterpol`. We are not aware of any publication describing the tool.

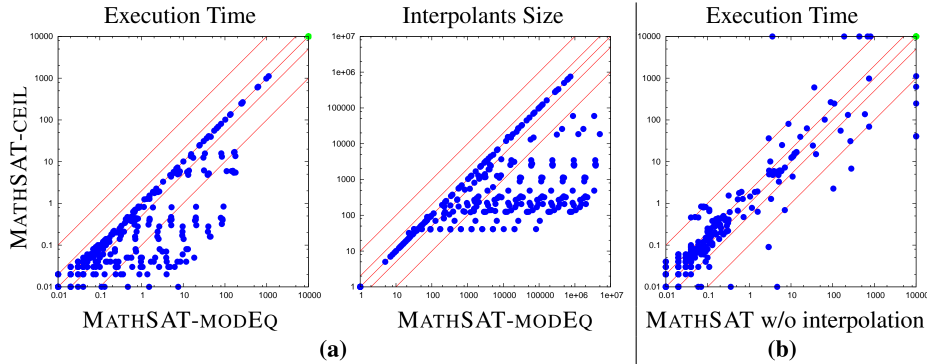

Fig. 4. (a) Comparison between MATHSAT-MODEQ and MATHSAT-CEIL configurations for interpolation. (b) Execution time overhead for interpolation with MATHSAT-CEIL.

is the same also when we compare the sizes of the produced interpolants, measured in number of nodes in a DAG representation of formulas. Comparisons on interpolant size are reported in Fig. 3, which shows that, on average, the interpolants produced by MATHSAT are comparable to those produced by other tools. In fact, there are some cases in which SMTINTERPOL produces significantly-smaller interpolants, but we remark that MATHSAT can solve 261 more instances than SMTINTERPOL.[13]

The differences between MATHSAT-MODEQ and MATHSAT-CEIL become evident when we compare the two configurations directly. The plots in Fig. 4(a) show that MATHSAT-CEIL is dramatically superior to MATHSAT-MODEQ, with gaps of up to two orders of magnitude in execution time, and up to four orders of magnitude in the size of interpolants. Such differences are solely due to the use of the ceiling function in the generated interpolants, which prevents the blow-up of the formula wrt. the size of the proof of unsatisfiability. Since most of the differences between the two configurations occur in benchmarks that none of the other tools could solve, the advantage of using ceilings was not visible in Figs. 2 and 3.

Finally, in Fig. 4(b) we compare the execution time of producing interpolants with MATHSAT-CEIL against the solving time of MATHSAT with interpolation turned off. The plot shows that the restriction on the kind of extended branch-and-bound lemmas generated when computing interpolants (see §3.3) can have a significant impact on individual benchmarks. However, on average MATHSAT-CEIL is not worse than the "regular" MATHSAT, and the two can solve the same number of instances, in approximately the same total execution time.

[13] The plots of Fig. 3 show also some apparently-strange outliers in the comparison with INTERPOLATINGOPENSMT. A closer analysis revealed that those are instances for which INTERPOLATINGOPENSMT was able to detect that the inconsistency of $A \wedge B$ was due solely to A or to B, and thus could produce a trivial interpolant $\bot$ or $\top$, whereas the proof of unsatisfiability produced by MATHSAT involved both A and B. An analogous situation is visible also in the comparison between MATHSAT and SMTINTERPOL, this time in favor of MATHSAT.

6 Conclusions

In this paper, we have presented a novel interpolation algorithm for $\mathcal{LA}(\mathbb{Z})$ that allows for producing interpolants from arbitrary cutting-plane proofs without the need of performing quantifier elimination. We have also shown how to exploit this algorithm, in combination with other existing techniques, in order to implement an efficient interpolation procedure on top of a state-of-the-art SMT($\mathcal{LA}(\mathbb{Z})$)-solver, with almost no overhead in search, and with up to orders of magnitude improvements – both in execution time and in formula size – wrt. existing techniques for computing interpolants from arbitrary cutting-plane proofs.

References

1. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on Demand in SAT Modulo Theories. In *LPAR'06*, volume 4246 of *LNCS*. Springer, 2006.
2. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 25. IOS Press, 2009.
3. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In *IJCAR*, volume 6173 of *LNCS*. Springer, 2010.
4. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. *ACM Trans. Comput. Logic*, 12(1), October 2010.
5. 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*. Springer, 2009.
6. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
7. A. Fuchs, A. Goel, J. Grundy, S. Krstic, and C. Tinelli. Ground interpolation for the theory of equality. In *TACAS'09*, volume 5505 of *LNCS*. Springer, 2009.
8. A. Goel, S. Krstic, and C. Tinelli. Ground Interpolation for Combined Theories. In *CADE-22*, volume 5663 of *LNCS*. Springer, 2009.
9. A. Griggio. A Practical Approach to SMT($\mathcal{LA}(\mathbb{Z})$). SMT 2010 Workshop, July 2010.
10. H. Jain, E. M. Clarke, and O. Grumberg. Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In *CAV'08*, volume 5123 of *LNCS*. Springer, 2008.
11. D. Kapur, R. Majumdar, and C. G. Zarba. Interpolation for data structures. In *FSE'05*. ACM, 2006.
12. D. Kroening, J. Leroux, and P. Rümmer. Interpolating Quantifier-Free Presburger Arithmetic. In *LPAR*, 2010. To Appear. Available at `http://www.philipp.ruemmer.org/publications.shtml`.
13. C. Lynch and Y. Tang. Interpolants for Linear Arithmetic in SMT. In *ATVA'08*, volume 5311 of *LNCS*. Springer, 2008.
14. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.
15. P. Pudlák. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symb. Logic*, 62(3), 1997.
16. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, 1991.
17. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. *J. Symb. Comput.*, 45(11), 2010.
18. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
19. G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE-20*, volume 3632 of *LNCS*. Springer, 2005.