Leveraging Linear and Mixed Integer Programming for SMT

Tim King* Clark Barrett* Cesare Tinelli†

*New York University †The University of Iowa

***Abstract*—SMT solvers combine SAT reasoning with specialized theory solvers either to find a feasible solution to a set of constraints or to prove that no such solution exists. Linear programming (LP) solvers come from the tradition of optimization, and are designed to find feasible solutions that are optimal with respect to some optimization function. Typical LP solvers are designed to solve large systems quickly using floating point arithmetic. Because floating point arithmetic is inexact, rounding errors can lead to incorrect results, making inexact solvers inappropriate for direct use in theorem proving. Previous efforts to leverage such solvers in the context of SMT have concluded that in addition to being potentially unsound, such solvers are too heavyweight to compete in the context of SMT. In this paper, we describe a technique for integrating LP solvers that improves the performance of SMT solvers without compromising correctness. These techniques have been implemented using the SMT solver CVC4 and the LP solver GLPK. Experiments show that this implementation outperforms other state-of-the-art SMT solvers on the QF_LRA SMT-LIB benchmarks and is competitive on the QF_LIA benchmarks.**

I. Introduction

Solvers for Satisfiability Modulo Theories (SMT) combine the ability of fast Boolean satisfiability (SAT) solvers to find solutions for complex propositional formulas with the ability of specialized *theory solvers* to find solutions to systems of constraints with respect to specific first order theories. SMT solvers excel in applications that require reasoning about non-trivial Boolean combinations of specific theory atoms.

Theory solvers for linear real and integer arithmetic are found in nearly every modern SMT solver, and are an essential building block for verification applications built on top of SMT. The best-performing arithmetic theory solvers are based on an algorithm that adapts the well-known simplex method to the SMT setting [1]. Because of their use in verification, SMT solvers typically use exact precision numeric representations internally in order to ensure that their calculations are correct and do not compromise the soundness of the overall system. For many typical SMT problems with significant Boolean structure (such as the majority found in the SMT-LIB benchmark library), this approach is sufficient, as the required theory reasoning is not too complex and the numbers involved in the internal calculations tend to stay relatively small. Moreover, such problems require tens or hundreds of thousands of calls to the theory solver. Thus, the theory solver's ability to incorporate new constraints quickly, to rapidly detect inconsistencies, and to backtrack efficiently, are far more important for overall efficiency than is the speed of the internal numerical calculations. However, there do exist problems for which this is not the case. If the internal simplex solver receives constraints that lead to large and dense linear systems, then using exact precision for the calculations required for the simplex search can overwhelm the solver.

The standard simplex algorithm finds a solution that is "best" according to some criteria. This is made mathematically explicit by adding a linear objective function that is to be maximized. The linear constraints combined with a linear objective are called *Linear Programs* (LPs), and systems that solve them are called *LP solvers*. Simplex-based LP solvers differ from SMT solvers in several important ways, including the following: (i) LP solvers solve only conjunctions of constraints - they cannot handle arbitrary Boolean combinations; (ii) LP solvers focus on both feasibility and optimization rather than just feasibility; (iii) LP solvers (generally) use floating point rather than exact precision arithmetic internally; and (iv) the product of many decades of research, modern LP solvers incorporate highly sophisticated techniques, making them very efficient in practice. The techniques used in LP solvers have been extended to the problem of optimizing constraints where all or some of the variables are required to be integers (Integer Programming (IP) and Mixed Integer Programming (MIP)).

On challenging simplex instances, LP and MIP solvers are considerably more efficient than the techniques used inside of SMT solvers. However, LP and MIP solvers are not optimized for rapid incremental calls, making them inefficient as theory solvers for many SMT applications. Also, their use of floating point means that they will occasionally return incorrect results. In this paper, we show how LP and MIP solvers can be efficiently and soundly incorporated into a modern SMT solver. Our work builds on previous efforts to leverage LP solvers for SMT but is the first to obtain significant improvements in performance by doing so. It is also the first to attempt integrating a MIP solver with SMT.

The rest of the paper is organized as follows. We give an overview of relevant background on SMT and simplex in Section II. Section III discusses our approach for integrating an LP solver in a theory solver for linear real arithmetic, and section IV shows how to extend this strategy to use an MIP solver in a theory solver for linear integer arithmetic. We conclude with section V, which reports and discusses experimental results.

II. Background

The core SMT problem is to determine whether a first order formula ϕ is satisfiable with respect to a fixed first order theory $\mathcal{T}$ [2]. Most modern SMT solvers rely on a DPLL($\mathcal{T}$) framework which combines a Boolean satisfiability (SAT) solver with decision procedures for various theories. The SAT solver is used to find an assignment of theory literals to truth values that is propositionally consistent with the Boolean skeleton of ϕ. The theory-specific modules used by SMT solvers are called *theory solvers*.

A theory solver for theory $\mathcal{T}$ takes as input a set Φ of theory literals[1] and determines whether Φ is consistent with respect to $\mathcal{T}$. If so, the theory solver responds with $\mathbf{Sat}$ and (optionally) a *model*, an assignment to the free variables in Φ that makes every formula in Φ true. If not, the theory solver responds with $\mathbf{Unsat}$ together with a small (ideally minimal) subset of Φ known to be unsatisfiable in $\mathcal{T}$, called a *conflict* set. A conflict set C is converted into a clause $\bigvee_{l \in C} \neg l$, and sent to the SAT solver. In addition, any $\mathcal{T}$-valid formula can be sent to the SAT solver by the theory solver and such formulas are called *theory lemmas*. Theory lemmas are used by theory solvers to help direct and guide the SAT solver during its search.

The focus of this paper is a novel theory solver for quantifier-free mixed linear integer and real arithmetic. We assume a language that includes the usual arithmetic constants and operators, a vector $\mathcal{V} = \langle x_1 \dots x_n \rangle$ of variables,[2] and a unary predicate $\texttt{IsInt}$. We assume that atoms are of the form (i) $\sum c_i \cdot x_i \bowtie d$ where c_i and d are rational constants, $\bowtie \in \{<, \leq, =\}$, and $x_i \in \mathcal{V}$, or of the form (ii) $\texttt{IsInt}(x_i)$, where $x_i \in \mathcal{V}$. An *assignment* a maps each $x_i \in \mathcal{V}$ to a value in the set $\mathbb{R}$ of real numbers. An assignment a *satisfies* an atom $\sum c_i \cdot x_i \bowtie d$ whenever $\sum c_i \cdot a(x_i) \bowtie d$ holds and satisfies $\texttt{IsInt}(x_i)$ whenever $a(x_i)$ is an integer. An atom that is satisfied by some assignment is said to be *satisfiable*. We lift the notion of satisfaction to arbitrary Boolean combinations of atoms in the natural way. We write $\phi \models_T \beta$ if every assignment satisfying ϕ also satisfies β. We assume that $\mathcal{V}$ is partitioned into a set $\mathcal{V}_{\mathbb{R}}$ of real variables and a set $\mathcal{V}_{\mathbb{Z}}$ of integer variables. The *integer-tightening* of a formula Φ is defined as $\phi \wedge \bigwedge_{z \in \mathcal{V}_{\mathbb{Z}}} \texttt{IsInt}(z)$, and the *real relaxation* of a formula α is obtained by replacing every application of the $\texttt{IsInt}$ predicate in α by $\texttt{True}$.[3] A formula is *integer-feasible* if its integer-tightening is satisfiable (and *integer-infeasible* otherwise), and an assignment is called *integer-compatible* if it assigns an integer to each integer variable. If a formula's real relaxation is satisfied by some assignment, we say it is *real-feasible* (or just *feasible*).

We first describe the well-known approach for simplex-based theory solvers in SMT, an approach we call *Simplex for DPLL($\mathcal{T}$)* (more details can be found in [1], [3]). The input is a conjunction of atoms of the form $\sum c_i \cdot x_i \leq d$. Weak inequalities are transformed by introducing a fresh real variable s for $\sum c_i \cdot x_i$ and rewriting the constraint as $s = \sum c_i \cdot x_i \wedge s \leq d$. The original x_i variables are called *structural* while the introduced s variables are called *auxiliary*. The orig function maps each auxiliary variable to its definition, $\mathrm{orig}(s) \equiv \sum c_i \cdot x_i$. Strict inequalities $\sum c_i \cdot x_i < d$ are rewritten as $\sum c_i \cdot x_i + \delta \leq d$ where δ is a small constant that can be determined later. To properly reason in the presence of δ, some of the internal constants are represented as special δ-*rationals*, pairs $\langle a, b \rangle$ of rationals interpreted as $a + b \cdot \delta$. Details on this technique can be found in [4].

After applying these transformations, the resulting constraints can always be written as: $T\mathcal{V} = 0 \wedge l \leq \mathcal{V} \leq u$, where T is a matrix, and l and u are vectors of lower and upper bounds on the variables. We use T_i to denote the i-th row of T. We use $l(x)$ and $u(x)$ to denote the lower and upper bound on a specific variable x. If x has no lower (upper) bound, then $l(x) = -\infty$ ($u(x) = +\infty$). The theory solver searches for an assignment $a : \mathcal{V} \mapsto \mathbb{R}$ that satisfies the constraints.

We assume T is an $n \times n$ matrix in *tableau form*: the variables $\mathcal{V}$ are partitioned into the *basic* variables $\mathcal{B}$ and *non-basic* variables $\mathcal{N}$ (to emphasize when a variable x_i is basic, we will write b_i as a synonym for x_i when $x_i \in \mathcal{B}$), and T_i is all zeroes iff $x_i \in \mathcal{N}$. Furthermore, for each column i such that $b_i \in \mathcal{B}$, we have $T_{k,i} = 0$ for all $k \neq i$ and $T_{i,i} = -1$. Thus, each nonzero row T_i of T represents a constraint $b_i = \sum_{x_j \in \mathcal{N}} T_{i,j} \cdot x_j$. Initially, the basic variables are exactly the auxiliary variables.

The simplex solver works by making a series of changes to an initial assignment a and the tableau T until the constraints are satisfied or determined to be unsatisfiable. During this process, $T \cdot a(\mathcal{V}) = 0$ is an invariant. To initially satisfy this invariant, we can set $a(x_i) = 0$ for all i. To maintain the invariant, whenever the assignment to a non-basic variable changes, the assignments to all dependent basic variables are also updated. Changes to the tableau are made via *pivoting*. Pivoting takes a basic variable b_i and a non-basic variable x_j such that $T_{i,j} \neq 0$, and swaps them: after pivoting, x_j becomes basic and b_i becomes non-basic.

Simplex for DPLL($\mathcal{T}$) solvers modify the assignment a and pivot the tableau T until a satisfying assignment is found or a *row conflict* is detected: a basic variable b_i violates one of its bounds but none of the non-basic variables that b_i depends on can be used to fix this without violating their own bounds. For example, suppose $a(b_i) > u(b_i)$ and for all $x_j \in \mathcal{N}$ with positive coefficients in row T_i ($T_{i,j} > 0$), $a(x_j) = l(x_j)$ and for all $x_k \in \mathcal{N}$ with negative coefficients in row T_i ($T_{i,k} < 0$), $a(x_k) = u(x_k)$. Then, $b_i \geq a(b_i)$ is entailed by the row and the constraints on the non-basic variables.[4] Since this contradicts $b_i \leq u(b_i)$, the entire system of constraints is unsatisfiable, and the following conflict set is generated:

$$\bigcup_{T_{i,j}>0} \{x_j \geq l(x_j)\} \cup \bigcup_{T_{i,k}<0} \{x_k \leq u(x_k)\} \cup \{b_i \leq u(b_i)\}.$$

[1]We will follow the common practice of overloading Φ to mean $\bigwedge_{\varphi \in \Phi} \varphi$ in contexts where a formula rather than a set is expected.

[2]For convenience, we will also use $\mathcal{V}$ to refer to the set $\{x_1 \dots x_n\}$.

[3]We assume that $\texttt{IsInt}$ occurs only positively in input formulas.

[4]There is a dual case when $a(b_i) < l(b_i)$.

The current best implementations of theory solvers for mixed linear integer and real arithmetic use a sound but incomplete procedure that layers integer reasoning on top of a solver for linear real arithmetic. Given a set Φ of atoms, the real solver is first used to solve the real relaxation of Φ. If the solver terminates, the result is either a conflict set or an assignment a (when Φ is real-feasible). In the first case, no additional work is necessary as a conflict set for the real relaxation of Φ is also a conflict set for Φ. In the second case, the assignment a is examined to see whether it is integer-compatible. If not, more work is needed to refine the assignment. The following *branching* technique can be used to ensure that the current assignment is refined in the next invocation of the theory solver: select a variable $x \in \mathcal{V}_{\mathbb{Z}}$ whose assignment is non-integer, and then send the following theory lemma to the SAT solver,

$$\texttt{IsInt}(x) \rightarrow (x \leq \lfloor a(x) \rfloor \vee x \geq \lceil a(x) \rceil) \tag{1}$$

The SAT solver will assert one of the two new bounds on x before reinvoking the theory solver.

Naive use of this heuristic can trigger an infinite sequence of branches, so more sophisticated methods based on *cutting planes* have been developed [5]. Consider a set Φ of assertions. A cutting plane is a plane through the solution space of the real relaxation of Φ that *cuts off* some of the non-integer-compatible assignments. More precisely, $\sum c_i x_i = d$ is a cutting plane for Φ and $\mathcal{H} \equiv \sum c_i x_i \leq d$ is a *cut* iff the following conditions hold: (i) every assignment satisfying the integer-tightening of Φ also satisfies $\mathcal{H}$; and (ii) at least one assignment satisfying the real relaxation of Φ also satisfies $\neg\mathcal{H}$.[5] The inequality $\mathcal{H}$ can be safely added to Φ without changing any of the (integer-compatible) satisfying assignments. A cut is always entailed by the integer-tightening of Φ and never by the real relaxation of Φ. Cuts can be implemented using theory lemmas, by sending the lemma $\Phi \Rightarrow \mathcal{H}$ to the SAT solver. Previous work has looked at using Gomory and Mixed Gomory cut techniques in SMT solvers [4].

III. Leveraging LP Solvers

The first contribution of this paper is a method for leveraging the strengths of both SMT and LP solvers to construct an efficient and robust theory solver for linear real arithmetic. This idea has been explored before. Early work by Yu and Malik [6] reports results on using an LP solver as a theory solver for SMT, but the issue of potentially incorrect results from the LP solver is not addressed. Faure et al. [7] integrate several LP solvers into the Barcelogic SMT solver [8]. They use an exact solver to lazily check the results from the LP solver to ensure soundness. Finally, in recent work by de Oliveira and Monniaux [9] (a continuation of the work in [10]), extensive experiments are done using an LP solver within OpenSMT [11]. In this work, the LP solver is called first and the results are used to "seed" the search in the exact solver.

[5]Often, an additional requirement is that $\mathcal{H}$ is not satisfied by the current assignment a. We will not require this here.

Thus most of the search is done by the LP solver, while the exact solver still ensures correctness.

In each of these studies, experimental results on SMT-LIB benchmarks show that existing SMT solvers outperform the experimental solvers modified to use LP solvers, even if the LP solver results are not checked for correctness. The main reason for this is that for these benchmarks (and the applications they represent), solving requires many related calls to the theory solver, each of which is relatively simple. The algorithms used in SMT solvers are optimized for this case and thus perform better, even though they use exact arithmetic which in general is much slower than floating point arithmetic. A solution to this problem advocated in [7] is to build a floating-point LP solver optimized for many, simple, related calls.

Here, we present an alternative approach. The idea is to take the two existing algorithms as they are and use each one only in cases when it is likely to do well. We thus use an exact solver optimized for fast incremental checks as the primary theory solver. However, we also instrument this solver so that it can detect when it is starting to have difficulty, and in these cases we have it call the LP solver.

The overall approach is given by the algorithm BalancedSolve shown in Figure 1. First, an efficient incremental exact solver ExactSolve is called with a heuristic cap on the number of pivots it may perform, k_{EX}. We assume that ExactSolve returns a status $\mathtt{c}$ (**Sat**, **Unsat**, or **Unknown**). If the exact solver returns **Sat** or **Unsat**, we are done and return the result. Otherwise, the heuristic cap was exceeded. In this case, the LP solver is called. We must convert the simplex problem described by T, l, and u to an analogous problem for the LP solver. We denote the LP analogs of the exact data by using the $\sim$ annotation. They are constructed (following [9]) as follows. For each auxiliary variable s, the equality $s = \text{orig}(s) \equiv \sum c_i x_i$, is added to $\widetilde{T}$ as $\widetilde{s} = \sum \text{float}(c_i) \cdot \widetilde{x_i}$, where the conversion function float maps a rational to the nearest float. For each variable $\widetilde{x}$, the bounds $\widetilde{l}(x)$ and $\widetilde{u}(x)$ are constructed from the *δ-rationals*, $l(x)$ and $u(x)$ by approximating δ as a small constant ϵ. For example, if $l(x) = \langle c, d \rangle$, then $\widetilde{l}(x)$ becomes $\text{float}(c + \epsilon \cdot d)$.

The LP solver is invoked with its own pivot limit k_{LP}. If the LP solver terminates with **Sat** or **Unsat**, we retrieve the assignment $\widetilde{a}$ as well as the final set of basic variables $\widetilde{\mathcal{B}}$ from the LP solver. The assignment $\widetilde{a}$ is converted into a rational assignment a' by the ImportAssignment routine (given below). The SeedExact procedure takes $\widetilde{\mathcal{B}}$ and a' and tries to verify the result of the LP solver using the exact solver. If this fails (or if the LP solver reaches its heuristic limit), the exact precision solver is run with a final limit k_{FI}. For *final* calls to BalancedSolve (i.e. the DPLL(T) SAT engine has found a propositionally satisfying assignment), k_{FI} should be $+\infty$. It can be less for non-final calls.

An important contribution of this paper is the procedure shown in Figure 2. This procedure attempts to assign a rational value to each variable that is close to the one given by the LP solver, but biased towards values that are easy to represent, partly because that makes them easier to calculate

```
1:  procedure BalancedSolve
2:      c ← ExactSolve($k_{EX}$)
3:      if c is Sat or Unsat then return c
4:      Construct $\widetilde{T}, \widetilde{l}, \widetilde{u}$ from $T, l, u$
5:      $\langle \widetilde{c}, \widetilde{a}, \widetilde{\mathcal{B}} \rangle$ ← LPSolve($k_{LP}, \widetilde{T}, \widetilde{l}, \widetilde{u}$)
6:      if $\widetilde{c}$ is Sat or Unsat then
7:          $a'$ ← ImportAssignment($\widetilde{a}$)
8:          c ← SeedExact($a', \widetilde{\mathcal{B}}$)
9:          if c is Sat or Unsat then return c
10:     return ExactSolve($k_{FI}$)
```

Fig. 1: The BalancedSolve procedure.

```
1:  procedure ImportAssignment($\widetilde{a}$)
2:      for all $x \in \mathcal{V}$ do
3:          $r$ ← DioApprox($\widetilde{a}(x), D$)
4:          if $|r - a(x)| \leq \epsilon$ then $r \leftarrow a(x)$
5:          if $x \in \mathcal{V}_{\mathbb{Z}}$ and $|r - \lfloor r \rceil| \leq \epsilon$ then $r \leftarrow \lfloor r \rceil$
6:          if $r > u(x)$ or $|r - u(x)| \leq \epsilon$ then $r \leftarrow u(x)$
7:          else if $r < l(x)$ or $|r - l(x)| \leq \epsilon$ then $r \leftarrow l(x)$
8:          $a'(x) \leftarrow r$
9:      return $a'$
```

Fig. 2: The ImportAssignment procedure.

```
1:  procedure SeedExact($a'$, $\widetilde{\mathcal{B}}$)
2:      for all $x \in \mathcal{N}$ do
3:          Update($x, a'(x) - a(x)$)
4:      $\mathcal{B}' \leftarrow \mathcal{N} \cap \widetilde{\mathcal{B}}$
5:      while $\mathcal{B}' \neq \emptyset$ do
6:          if $T$ has a row conflict then return Unsat
7:          if all variables satisfy their bounds then return Sat
8:          if $\exists\, i\, j.\ x_j \in \mathcal{B}' \wedge x_i \notin \widetilde{\mathcal{B}} \wedge T_{i,j} \neq 0$ then
9:              Pivot($i, j$)
10:             Update($i, a'(x_i) - a(x_i)$)
11:             $\mathcal{B}' \leftarrow \mathcal{B}' \setminus \{x_j\}$
12:         else return Unknown
13:     return Unknown
```

Fig. 3: The SeedExact procedure.

```
1:  procedure IntegerSolve
2:      c ← BalancedSolve()
3:      if c is Unsat then return c
4:      Construct $\widetilde{T}, \widetilde{l}, \widetilde{u}$ from $T, l, u$
5:      $\langle \widetilde{c}, \widetilde{a}, \widetilde{\mathcal{B}}, \widetilde{t} \rangle$ ← MIPSolve($k_{\text{MIP}}, \widetilde{T}, \widetilde{l}, \widetilde{u}$)
6:      if $\widetilde{c}$ is Unsat then c ← Replay($\widetilde{t}$)
7:      else if $\widetilde{c}$ is Sat then
8:          $a'$ ← ImportAssignment($\widetilde{a}$)
9:          c ← SeedExact($a', \widetilde{\mathcal{B}}$)
10:         if c is Unknown then c ← ExactSolve($+\infty$)
11:     if (c is Sat and $a$ is integer-compatible) or
            (c is Unsat) then return c
12:     Generate a branching theory lemma using (1)
13:     return Unknown
```

Fig. 4: The IntegerSolve procedure.

with, but also partly because the discarded portion often corresponds exactly to a rounding error. For each variable x in the assignment, ImportAssignment first approximates $\widetilde{a}(x)$ as a rational using a technique based on continued fraction expansion called Diophantine approximation [5]. This technique finds the closest rational value with a denominator less than some fixed constant integer D. Next, we check to see if this value is within ϵ of the last known assignment for x in the exact solver. If so, the last known assignment is used. Next, if $x \in \mathcal{V}_{\mathbb{Z}}$ and the value is within ϵ of an integer z ($\lfloor r \rceil$ denotes the nearest integer to r), then z is used. Finally, ImportAssignment examines the value with respect to $l(x)$ and $u(x)$. If the value violates one of these bounds or is within ϵ of a bound, then the bound is used instead.

The SeedExact routine (Fig. 3) attempts to duplicate the results from the LP solver within the exact solver. First the procedure updates the exact solver assignment by calling Update on each non-basic variable. Next it computes the set, $\mathcal{B}'$, of variables that are non-basic in the exact solver but were marked as basic by the LP solver. We loop until as many variables in $\mathcal{B}'$ as possible have been pivoted to become basic. At the beginning of each iteration, we visit all the rows of T to check for conflicts. ([3] discusses doing this check efficiently.) While checking for conflicts, we can also detect whether any basic variable violates its upper or lower bound. If not, we have a satisfying assignment and stop early. If neither check applies, we search for a pair of variables x_i, x_j such that x_j is in $\mathcal{B}'$ meaning it is non-basic but should be basic, and $T_{i,j} \neq 0$ and $x_i \notin \widetilde{\mathcal{B}}$ meaning that x_i is basic but should be non-basic. If we can find such a pair, we pivot i and j and update the assignment of x_i to $a'(x_i)$. Approximations made by the LP solver or by ImportAssignment mean that SeedExact may fail to detect a satisfying assignment or a conflict in which case it returns **Unknown**. The SeedExact procedure can be seen as achieving a similar effect as ForcedPivot in [10] using rounds of the simplex algorithm in [1].

An alternative to verifying the LP solution would be to use an exact external LP solver (e.g. [12]–[14]). However, the use of an exact external solver (as well as an attempt to implement their rather sophisticated techniques) is beyond the scope of this work. Our goal, rather, is to make a first effort at an efficient integration of *inexact* floating-point solvers within SMT search. Integrating an exact external solver would be an interesting direction for future work.

IV. Using MIP Solvers to Improve Theory Solvers for Mixed Linear Integer and Real Arithmetic

We show how to extend the technique from the previous section to mixed linear integer and real arithmetic. The IntegerSolve algorithm (Fig. 4) illustrates our approach. First, the real relaxation of the problem is solved using the BalancedSolve algorithm described above. If the real

PROPAGATE

$$\frac{\tilde{E} \subseteq C_N \cup \tilde{P} \quad \tilde{h} \text{ is an inequality constraint} \quad \tilde{E} \cup I \models_T \tilde{h}}{N_1 := N \cdot \langle \tilde{h}, \tilde{E} \rangle}$$

BRANCH

$$\frac{\tilde{a} \text{ satisfies } \tilde{P} \wedge C_N \quad v \in \mathcal{V}_\mathbb{Z} \quad \tilde{a}(v) = \alpha \quad \alpha \notin \mathbb{Z}}{N_1 := N \cdot \langle v \leq \lfloor \alpha \rfloor, \emptyset \rangle \quad \| \quad N_2 := N \cdot \langle v \geq \lceil \alpha \rceil, \emptyset \rangle}$$

Fig. 5: Derivation rules. N is the parent node, N_1 and N_2 its child nodes. The symbol $\cdot$ denotes sequence concatenation.

relaxation is unsatisfiable, then we are done. Otherwise, we construct an MIP instance and call an MIP solver (with a pivot limit k_{MIP}) to search for an integer-compatible solution. When **Unsat** is returned, we also retrieve a *proof tree* $\tilde{t}$, which is a record of the steps taken by the MIP solver, and attempt to verify the tree by *replaying* its proof in the exact solver using the REPLAY procedure described below. Otherwise, if **Sat** is returned, we attempt to verify the assignment as before. If the verification fails, we again call EXACTSOLVE to ensure that we have a solution to the real relaxation before continuing. If we are unable to verify that the problem is **Unsat** or do not find an integer-compatible assignment, we force a branch by generating a theory lemma of the form (1) and return.

We now show how proof trees extracted from the MIP solver can be replayed within the exact solver. For the rest of the section, let M be an MIP instance consisting of an LP problem P of the form $T\mathcal{V} = 0 \wedge l \leq \mathcal{V} \leq u$ with the integer-tightening constraints $I \equiv \bigwedge_{z \in \mathcal{V}_\mathbb{Z}} \texttt{IsInt}(z)$. Let $\tilde{P}$ be the approximate version of P obtained by converting all rational constants in P to their corresponding floating point constants.

The process that an MIP solver goes through before concluding that $\tilde{P}$ is integer-infeasible can be described at an abstract level as a search tree. The root node represents the initial problem $\tilde{P}$ and each non-root node is derived from its parent by adding either a cut or a branch to the problem. The leaves of the tree represent real-infeasible problems.

Formally, we define a tree node N as a sequence of pairs $\langle \tilde{h}, \tilde{E} \rangle$, where $\tilde{h}$ is an inequality constraint and $\tilde{E}$ is an *explanation*, a (possibly empty) finite set, each element of which is either some $\tilde{h}'$ where $\langle \tilde{h}', \tilde{E}' \rangle$ appears earlier in N or is a constraint from the initial problem $\tilde{P}$. We denote by C_N the set $\{\tilde{h} \mid \langle \tilde{h}, \tilde{E} \rangle \in N\}$.

The root node of a proof tree is the empty sequence. Each non-root node is the result of applying to its parent node one of the derivation rules in Figure 5. The PROPAGATE rule is used to record when the MIP solver adds a cut. The cut must be entailed by some subset of constraints in the current MIP problem. The cut and its explanation are recorded in the child sequence. The BRANCH rule is used to record when the MIP solver does a case split on an integer variable. This can happen when the MIP solver has a solution $\tilde{a}$ to the real relaxation of the current problem that is not integer-compatible. The MIP solver chooses an integer variable v that has been assigned a real value α and enforces the constraint $v \leq \lfloor \alpha \rfloor \vee v \geq \lceil \alpha \rceil$. The rule has two children, each of which

```
1:  procedure REPLAY($H, t$)
2:    $C_H \leftarrow \{h | \langle h, E \rangle \in H\}$
3:    if $t$ is a is a leaf node $N$ then
4:      Construct $T$, $l$, $u$ from $P \cup C_H$
5:      $\mathsf{c} \leftarrow$ BALANCEDSOLVE()
6:      if $\mathsf{c}$ is not Unsat then return Unknown
7:      Let $\psi \subseteq P \cup C_H$ be the conflict from BALANCEDSOLVE
8:      return REGRESS($\psi, H$)
9:    if the root of $t$ has only one child $c$ then
10:     $t' \leftarrow$ subtree of $t$ rooted at $c$
11:     $\langle h, E \rangle \leftarrow$ IMPORTCONSTRAINT($last(c)$)
12:     if $E \subseteq C_H \cup P$ and $E \cup I \models_T h$ then
13:       return REPLAY($H \cdot \langle h, E \rangle, t'$)
14:     else return REPLAY($H, t'$)
15:   if the root of $t$ has two children $c_1$ and $c_2$ then
16:     for $i = 1, 2$ do
17:       $t_i \leftarrow$ subtree of $t$ rooted at $c_i$
18:       $\langle h_i, \emptyset \rangle \leftarrow last(c_i)$
19:       $K_i \leftarrow$ REPLAY($H \cdot \langle h_i, \emptyset \rangle, t_i$)
20:     $K \leftarrow$ RESOLVEBRANCH($K_1, K_2$)
21:     return REGRESS($K, H$)
```

Fig. 6: The REPLAY procedure.

records in its sequence one of the two branch cases (with an empty explanation). A node N is a leaf when the MIP solver concludes that the problem $\tilde{P} \cup C_N$ is (real)-infeasible.

Ideally, a proof tree would allow us to prove that the original problem P is integer-infeasible. However, because of the approximate representation used by the MIP solver, this is not always the case. As a consequence, our theory solver uses the proof tree just as a guide for its own internal attempt to prove that P is integer-infeasible. This process is captured at a high level by the REPLAY function.

The REPLAY function is shown in Figure 6. It takes an initially empty sequence H and a proof tree t, and traverses the tree with the goal of computing a *conflict*, a subset of the constraints in the original LP problem P that are integer-infeasible. As REPLAY traverses the tree, it constructs a sequence H which is analogous to the sequences in the tree nodes, except that it contains only those constraints that the internal exact solver has successfully replayed and so may only be a subset of those in the tree node. (The REPLAY procedure returns **Unknown** if the replay has failed.)

If t is a leaf node, then $\tilde{P} \cup C_N$ should be integer-infeasible. We check the exact analog, $P \cup C_H$. If unsuccessful, we fail, returning **Unknown**; otherwise, we return a conflict. To compute the conflict, we make use of an auxiliary function, REGRESS, which is not shown. REGRESS takes a conflict K and a sequence H of constraint-explanation pairs and recursively replaces any constraint in K by explanation [assuming the explanation is non-empty]. The net effect is to ensure a conflict which does not contain derived cuts.

If the root of t has a single child, this child must have been derived using the PROPAGATE rule. The last element of the sequence in the child node represents the new cut and its explanation. We convert the cut and its explanation to their exact analogs and then verify that we can derive the cut h from the exact constraints in E. These steps are explained in more detail below. If the cut can be verified, it and its explanation are included in the parameter H passed to the next recursive call to REPLAY. If not, the recursive call is made without h in the hopes that it is not needed to derive a conflict.

The final case is when the root of t has two children, indicating that the BRANCH rule was applied. Because branch constraints only use integers, importing them cannot fail. We are always able to represent them exactly. Thus, we simply call REPLAY recursively on each of the two sub-trees, passing one of the branch conditions to each sub-tree. The RESOLVE-BRANCH procedure constructs a conflict from the two returned conflicts K_1 and K_2. The procedure returns either: (i) the result of resolving K_1 and K_2 to remove the branch literals, (ii) K_i if it does not involve the branch, or (iii) **Unknown** otherwise. (The failure case requires at least one branch to be unknown.) As before, we use REGRESS to ultimately construct a conflict with constraints in P (we require REGRESS to return **Unknown** if K is **Unknown**).

Lines 12 and 13 of REPLAY require converting $\langle \widetilde{h}, \widetilde{E} \rangle$ to an exact analog, $\langle h, E \rangle$, and then verifying that h can be derived from E. We have implemented support for both Mixed-Gomory cuts and a variant of aggregated Mixed Integer Rounding cuts [15]. We will only explain here how reconstruction works for a special case of *Gomory cutting planes*.

The MIP solver can add a Gomory cutting plane $\widetilde{h}$ when the following conditions hold: (i) there is a row in $\widetilde{T}$, $b_i = \sum \widetilde{T_{i,j}} \cdot x_j$; (ii) all of the non-basic variables on the row are assigned to either their upper or lower bound; (iii) a subset of the variables on the row, that must include the basic variable b_i, are integer variables; and (iv) the assignment of b_i is non-integer. The premises (i)-(iv) make up the explanation $\widetilde{E}$.[6] For simplicity of presentation, we additionally assume all of the variables are integer and all the coefficients $\widetilde{T_{i,j}}$ are positive and assigned to their upper bounds. The assignment to b_i is then determined by the upper bounds of the non-basic variables, $\widetilde{a}(b_i) = \sum \widetilde{T_{i,j}} \cdot \widetilde{u}(x_j)$. The cut $\widetilde{h}$ for these constraints is then

$$\sum \frac{\widetilde{T_{i,j}}}{\widetilde{a}(b_i) - \lfloor \widetilde{a}(b_i) \rfloor} \left(\widetilde{u}(x_j) - x_j \right) \geq 1.$$

Given $\langle \widetilde{h}, \widetilde{E} \rangle$, we can attempt to derive a trusted cut and explanation $\langle h, E \rangle$ as follows. To reconstruct the cut, for every bound $x_j \leq \widetilde{u}(x_j) \in \widetilde{E}$, there must be a corresponding bound $x_j \leq u(x_j)$ in the exact system. (Note: $x_j \leq u(x_j)$ can be in either P or C_H.) Next we attempt to reconstruct the row $b_i = \sum \widetilde{T_{i,j}} x_j$ in exact precision as a row vector α. The coefficient for the basic variable in α is -1 ($\alpha_i = -1$). Nonbasic variables' coefficients are estimated from the approximate variables, $\alpha_j = \text{DIOAPPROX}(\widetilde{T_{i,j}}, D)$. If after approximation, the sign of α_j does not match the sign of $\widetilde{T_{i,j}}$, this cut cannot be reproduced. The equalities $T\mathcal{V} = 0$ entail $\sum \alpha_k x_k = 0$ iff α is in the row span of T. This entailment can be checked by replacing auxiliary variables with their original definitions,

$$\alpha_i \cdot x_i + \sum_{x_j \text{is structural}} \alpha_j \cdot x_j + \sum_{x_k \text{is auxiliary}} \alpha_k \cdot \text{orig}(x_k),$$

and rejecting this cut if any of the coefficients do not cancel to 0.[7] The row α and the bounds $u(x_j)$ are used to generate $b = \sum \alpha_j \cdot u_j$, which can be thought of as a potential assignment to b_i. The cut cannot be reproduced if $b \in \mathbb{Z}$. If the value of b is non-integer, the Gomory cut h

$$h : \sum \frac{\alpha_j}{b - \lfloor b \rfloor} (u(x_j) - x_j) \geq 1$$

has been reproduced in exact precision. The explanation for h, E, includes the upper bounds $x_j \leq u(x_j)$, the integer constraints, and the equations $x_k = \text{orig}(x_k)$.

V. EXPERIMENTS AND DISCUSSION

All of the algorithms in this paper have been implemented in the CVC4 SMT solver [16].[8] In this section, we report the results of experiments using these implementations.

The implementation contains additional heuristics and several tunable parameters. While the authors have not done a formal tuning of any of these parameters, we include these values for completeness. There are two different simplex implementations in CVC4, one that follows the well-known simplex adapted for SMT described in [1], [4], and one based on sum-of-infeasibilities as described in [3]. The experiments were run using the latter method for the EXACTSOLVE procedure with a pivot cap of $k_{EX} = 200$ in Fig. 1 (with $k_{FI} = 200$ for non-final calls). Values of other parameters used in our experiments are $D = 2^{26}$; $\epsilon = 10^{-9}$; $k_{LP} = 10000$; and $k_{\text{MIP}} = 200000$. For both the LP and MIP solvers, we use the floating-point simplex solver in GLPK version 4.52 [17], instrumented to communicate the additional information needed by CVC4 in order to verify assignments, conflicts, and proof trees.[9] To avoid branching loops in GLPK, GLPK is halted if it branches 100 times on any one variable. To keep the size of the numeric constants manageable, we reject any cut containing a coefficient $\frac{n}{d}$ where $\log_2(|n|) + \log_2(|d|) > 512$. Further, we have a heuristic that dynamically disables the GLPK solver if it claims the problem is real-feasible and then integer-infeasible without generating any branches or cuts, a strange situation that happens with the `convert` benchmarks (see discussion below for details). GLPK is also dynamically disabled if CVC4's bignum package throws an exception while trying to import a floating point number. CVC4 has a heuristic that automatically detects and reencodes benchmarks in the `QF_LRA` family miplib (which are derived from benchmarks in [18]) in something closer to their original form.[10]

[6] See [4] for a Gomory cutting plane rule without additional assumptions.

[7] α can also be generated by Gaussian elimination from $\bigwedge x_k = \text{orig}(x_k)$.

[8] Experiments were run using a branch of CVC4 available at github.com/timothy-king/CVC4/CVC4 (commit 2550b6d).

[9] Source for this modified version of GLPK is available at github.com/timothy-king/glpk-cut-log (commit a35b8e).

[10] A comparison of other solvers on the miplib problems after this reencoding gave similar results to those reported in Table I.

			CVC4+MIP		CVC4		yices2		mathsat5		Z3		altergo		cutsat		scip		glpk	
set	# inst.	# sel.	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
Selecting all benchmarks in the family																				
QF_LRA	652	652	645	6966	636	8557	632	5350	622	10913	615	5696	-	-	-	-	-	-	-	-
non-conj. QF_LIA	4579	4579	4489	86854	4472	86375	4375	30656	4543	55417	4474	75171	3956	262031	-	-	-	-	-	-
conj. QF_LIA	1303	1303	1249	11130	1068	31054	1111	55691	1154	33260	1039	19015	1232	2055	1018	35330	1255	7164	1173	8895
total	6534	6534	6383	104950	6176	125986	6118	91697	6319	99590	6128	99882	-	-	-	-	-	-	-	-
Selecting QF_LRA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
miplib	42	37	30	1530	21	3037	23	2730	17	5682	18	2435	-	-	-	-	-	-	-	-
DTP-Scheduling	91	4	4	4	4	4	4	0	4	2	4	1	-	-	-	-	-	-	-	-
latendresse	18	18	18	767	18	836	12	85	10	99	0	0	-	-	-	-	-	-	-	-
total	-	59	52	2301	43	3877	39	2815	31	5783	22	2436	-	-	-	-	-	-	-	-
Selecting non-conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
convert	319	282	208	9646	193	9343	188	4337	274	1876	282	118	166	272	-	-	-	-	-	-
bofill-scheduling	652	460	460	5401	458	4490	460	748	460	1519	460	2060	67	55	-	-	-	-	-	-
CIRC	51	11	11	0	11	0	11	0	11	0	11	0	11	0	-	-	-	-	-	-
calypto	37	37	37	3	37	3	37	0	37	6	36	5	35	24	-	-	-	-	-	-
nec-smt	2780	207	207	17276	207	18045	199	777	207	17925	201	7209	184	23724	-	-	-	-	-	-
wisa	5	1	1	0	1	0	1	0	1	1	1	0	1	0	-	-	-	-	-	-
total	-	998	924	32326	907	31881	896	5862	990	21327	991	9392	464	24075	-	-	-	-	-	-
Selecting conjunctive QF_LIA benchmarks on which either LPSOLVE or MIPSOLVE was called at least once																				
dillig	233	189	189	49	157	9823	175	8557	188	7185	166	1269	189	5	166	5840	189	42	189	3
miplib2003	16	8	4	307	4	1283	4	507	5	354	5	1089	0	0	6	146	7	17	6	295
prime-cone	37	37	37	2	37	2	37	2	37	1	37	2	37	1	37	4	37	1	37	0
slacks	233	188	166	61	93	2003	107	15672	119	4741	90	1994	188	84	96	6324	161	2361	101	11
CAV_2009	591	424	424	69	346	10035	376	26351	421	10236	354	2759	423	323	377	17015	424	105	424	6
cut_lemmas	93	74	62	9581	64	6865	72	1662	45	9472	38	5858	74	267	15	1887	72	1757	71	760
total	-	920	882	10069	701	30011	771	52751	815	31989	690	12971	911	680	697	31216	890	4283	828	1075

TABLE I: Experimental results on QF_LRA and QF_LIA benchmarks.

The experiments were conducted on the StarExec platform [19] with a CPU time limit of 1500 seconds and a memory limit of 8GB. The first segment of Table I compares our implementation with other SMT solvers over the full sets of QF_LRA and QF_LIA benchmarks from the SMT-LIB library (the "2013-03-07" version on StarExec), extended with the latendresse QF_LRA benchmarks from [3]. The QF_LIA benchmarks are divided into the *conjunctive* subset and the non-conjunctive subset. The conjunctive subset consists of all families, all of whose benchmarks are a simple conjunction of constraints.[11] The primary experimental comparison is between a configuration of CVC4 running just its internal solvers ("CVC") against a configuration with the techniques of this paper enabled ("CVC4+MIP"). We additionally compare with similar state-of-the-art SMT solvers: mathsat5 (smtcomp12 version) [20], z3 (v4.3.1) [21], and yices2 (v2.2.0) [22]. We include a comparison against the version of AltErgo [23] used in [24] on just the QF_LIA benchmarks. For the conjunctive subset, we also give results for several solvers that support only conjunctive benchmarks: cutsat (CADE11) [25], SCIP (scip-3.0.0-ex+spx) [13], [26], and glpk (4.52) [17]. This version of SCIP handles MIP problems in exact precision.

The remaining segments of Table I give more detailed results for QF_LRA benchmarks, non-conjunctive QF_LIA benchmarks, and conjunctive QF_LIA benchmarks respectively. In each segment, we report only the results on benchmarks for which CVC4+MIP invokes GLPK at least once. (For each family, the second column of numbers indicates how many benchmarks in the family are included in the results. See cs.nyu.edu/~taking/fmcad14_selections for a list of selected benchmarks.)

[11]The conjunctive families are dillig, miplib2003, prime-cone, slacks, CAV_2009, cut_lemmas, pidgeons, and pb2010. We translated these into the SMT-LIBv1.0 and MPS formats: cs.nyu.edu/~taking/conjunctive_integers.tbz.

			Sat		**Unsat**	
set	# sel.	MIPSOLVE calls	attempts	successes	attempts	successes
QF_LIA	1393	3873	2559	1058	652	425
convert	208	2130	1356	1	178	3
bofill-scheduling	254	254	245	245	0	0
CIRC	11	85	6	5	79	77
calypto	37	375	77	23	293	278
wisa	1	1	1	1	0	0
dillig	189	228	225	185	3	2
miplib2003	4	10	3	3	5	4
prime-cone	37	37	19	19	18	18
slacks	166	195	168	162	3	3
CAV_2009	424	469	459	414	8	7
cut_lemmas	62	89	0	0	65	33

TABLE II: Success rate of reproducing results of MIPSOLVE

To better understand how successful the verification and replaying algorithms for integers described in Section IV are, we analyzed all of the `QF_LIA` instances which were solved by CVC4+MIP and for which MIPSolve was invoked at least once, and collected the following statistics: the number of times MIPSolve was called, the number of attempts and successes at verifying **Sat** results from MIPSolve, and the number of attempts and successes at replaying **Unsat** results from MIPSolve. The results are shown in Table II.

On `QF_LRA` benchmarks, CVC4+MIP solves all of the problem instances that the already competitive CVC4 does plus 9 additional problems (solving more than any other solver), all from the challenging miplib family. After preprocessing, these benchmarks are represented internally as mixed linear real and integer problems, so the IntegerSolve procedure is invoked. CVC4+MIP is the only solver to solve the `opt1217--{27,37,57}.smt2` benchmarks, and it does so in about 1s each. These and a handful of other miplib problems are real-infeasible and are solved very quickly by BalancedSolve. IntegerSolve is able to verify that several other miplib benchmarks are **Sat**. It was not able to successfully solve the most difficult problems which are real-feasible but integer-infeasible.

CVC4+MIP is also quite competitive on the `QF_LIA` problem instances. Particularly dramatic is the improvement of CVC4+MIP over CVC4 on the (related) families dillig, slacks, and CAV_2009_benchmarks. These benchmarks are small, randomly generated, conjunctive problems that are mostly satisfiable [25], [27]. It appears from Table II that CVC4+MIP does well on these families due to a high proportion of successes when ImportAssignment and SeedExact are used to verify **Sat** instances. Excluding the convert family, GLPK returned **Sat** 1203 times, and in 1057 cases, we were able to verify this with the exact solver. Given the challenges of implementing branching and cutting within SMT solvers, this suggests that the technique of soundly verifying results from an external solver offers a new powerful tool in designing `QF_LIA` solvers. The empirical results on the Replay procedure, while not as dramatic, are also promising. Excluding the convert benchmarks, Replay was successful on 425 out of 652 invocations and did particularly well on (relatively) easy benchmarks e.g. calypto and prime-cone.

CVC4+MIP is competitive with the dedicated conjunctive solvers we included. Of course, its performance is limited by that of GLPK (Interestingly, CVC4+MIP outperforms GLPK on these benchmarks.) Though most of the improvement of CVC4+MIP over CVC4 is on conjunctive benchmarks, the authors suspect this to be an artifact of the benchmarks.

The convert family is interesting in that almost every proof reported by GLPK on these benchmarks fails to replay. The benchmarks contain integer equalities between variables with coefficients of massively different scales. To ensure numerical stability, GLPK increases each bound by some amount ϵ, where ϵ is proportional to the size of the bound. Because of the dramatic differences of scale in the coefficients in the convert family, GLPK increases some bounds by a large amount and others by a small amount. As a result, GLPK frequently makes incorrect conclusions (both feasible and infeasible) about subproblems from this family. These benchmarks thus present a challenge for the techniques given in section IV and are a good subject for future research.

Acknowledgments: We would like to thank Morgan Deters for his help running experiments and Bruno Dutertre for providing us with a custom version of yices2.

References

[1] B. Dutertre and L. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, 2006, pp. 81–94.
[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Satisfiability*. IOS, 2009, ch. 26.
[3] T. King, C. Barrett, and B. Dutertre, "Simplex with sum of infeasibilities for SMT," in *FMCAD*, 2013, pp. 189–196.
[4] B. Dutertre and L. de Moura, "Integrating Simplex with DPLL(T)," SRI International, Tech. Rep. SRI-CSL-06-01, 2006.
[5] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York, NY, USA: Wiley-Interscience, 1988.
[6] Y. Yu and S. Malik, "Lemma Learning in SMT on Linear Constraints," in *SAT*, 2006, pp. 142–155.
[7] G. Faure, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers," in *SAT*, 2008, pp. 77–90.
[8] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodrguez-Carbonell, and A. Rubio, "The Barcelogic SMT solver," in *CAV*, 2008, pp. 294–298.
[9] D.C.B. de Oliveira and D.Monniaux, "Experiments on the feasibility of using a floating-point simplex in an SMT solver," in *PAAR*.CEUR, 2012.
[10] D. Monniaux, "On using floating-point computations to help an exact linear arithmetic decision procedure," in *CAV*, 2009, pp. 570–583.
[11] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *TACAS*, 2011, pp. 150–153.
[12] D. L. Applegate, W. Cook, S. Dash, and D. G. Espinoza, "Exact solutions to linear programming problems," *Operations Research Letters*, vol. 35, no. 6, pp. 693 – 699, 2007.
[13] W. Cook, T. Koch, D. E. Steffy, and K. Wolter, "A hybrid branch-and-bound approach for exact rational mixed-integer programming," *Math. Program. Comput.*, vol. 5, no. 3, pp. 305–344, 2013.
[14] A. Neumaier and O. Shcherbina, "Safe bounds in linear and mixed-integer linear programming," *Mathematical Programming*, vol. 99, no. 2, pp. 283–296, 2004.
[15] H. Marchand and L. A. Wolsey, "Aggregation and mixed integer rounding to solve mips," *Operations Research*, vol. 49, no. 3, pp. 363–371, 2001.
[16] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*, 2011, pp. 171–177.
[17] A. Makhorin, "GNU Linear Programming Kit, Version 4.52," jun 2012. [Online]. Available: http://www.gnu.org/software/glpk/glpk.html
[18] T. Achterberg, T. Koch, and A. Martin, "Miplib 2003," *Operations Research Letters*, vol. 34, no. 4, pp. 361 – 372, 2006.
[19] A. Stump, G. Sutcliffe, and C. Tinelli, "StarExec: a Cross-Community Infrastructure for Logic Solving," in *IJCAR*, 2014, pp. 367–373.
[20] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS*, 2013.
[21] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, 2008, pp. 337–340.
[22] B. Dutertre, "Yices 2.2," in *CAV*, 2014, pp. 737–744.
[23] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout, "The Alt-Ergo Automated Theorem Prover." [Online]. Available: http://alt-ergo.lri.fr
[24] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, A. Mahboubi, A. Mebsout, and G. Melquiond, "A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic," in *IJCAR*, 2012, pp. 67–81.
[25] D. Jovanovic and L. M. de Moura, "Cutting to the Chase Solving Linear Integer Arithmetic," in *CADE*, 2011, pp. 338–353.
[26] T. Achterberg, "Scip: solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.
[27] I. Dillig, T. Dillig, and A. Aiken, "Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers," in *CAV*, 2009, pp. 233–247.