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 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 $\phi$ is satisfiable with respect to a fixed first order theory $T$ [2]. Most modern SMT solvers rely on a DPLL($T$) 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 $\phi$. The theory-specific modules used by SMT solvers are called theory solvers.

A theory solver for theory $T$ takes as input a set $\Phi$ of theory literals and determines whether $\Phi$ is consistent with respect to $T$. If so, the theory solver responds with Sat and (optionally) a model, an assignment to the free variables in $\Phi$ that makes every formula in $\Phi$ true. If not, the theory solver responds with Unsat together with a small (ideally minimal) subset of $\Phi$ known to be unsatisfiable in $T$, called a conflict set. A conflict set $C$ is converted into a clause $\bigvee_{i \in C} \neg l_i$, and sent to the SAT solver. In addition, any $T$-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 order theory. We first describe the well-known approach for simplex-based theory solvers in SMT, an approach we call Simplex for DPLL($T$) [1, 3]. The input is a conjunction of atoms of the form $c_i \cdot x_i \leq d$. Weak inequalities are transformed by introducing a fresh real variable $s$ for $\sum c_i \cdot x_i$ and rewriting the constraint as $s = \sum c_i \cdot x_i \land s \leq d$. 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, orig($s$) $\equiv \sum c_i \cdot x_i$. Strict inequalities $\sum c_i \cdot x_i < d$ are rewritten as $\sum c_i \cdot x_i + \delta < d$ where $\delta$ is a small constant that can be determined later. To properly reason in the presence of $\delta$, some of the internal constants are represented as special $\delta$-rationals, pairs $(a, b)$ 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: $T \lor 0 \land l \leq v \leq u$, 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 $a : V \rightarrow \mathbb{R}$ that satisfies the constraints.

We assume $T$ is an $n \times n$ matrix in tableau form: the variables $V$ are partitioned into the basic variables $B$ and non-basic variables $N$ (to emphasize when a variable $x_i$ is basic, we will write $b_i$ as a synonym for $x_i$ when $x_i \in B$), and $T_i$ is all zeroes iff $x_i \in N$. Furthermore, for each column $i$ such that $b_i \in B$, 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 $b_i = \sum_{x_j \in N} T_{i,j} \cdot x_j$. 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, $T \cdot a(V) = 0$ 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($T$) 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 $x_j \in N$ with positive coefficients in row $T_i$ ($T_{i,j} > 0$), $a(x_j) = l(x_j)$ and for all $x_k \in N$ 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. Since this contradicts $b_i \leq u(b_i)$, the entire system of constraints is unsatisfiable, and the following conflict set is generated:

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

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

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

3We assume that IsInt occurs only positively in input formulas.

4There 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 \( \Phi \) of atoms, the real solver is first used to solve the real relaxation of \( \Phi \). If the solver terminates, the result is either a conflict set or an assignment \( a \) (when \( \Phi \) is real-feasible). In the first case, no additional work is necessary as a conflict set for the real relaxation of \( \Phi \) is also a conflict set for \( \Phi \). 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 \( x \in V_\mathbb{Z} \) whose assignment is non-integer, and then send the following theory lemma to the SAT solver,

\[
\text{IsInt}(x) \rightarrow (x \leq \lfloor a(x) \rfloor \lor x \geq \lceil a(x) \rceil)
\]  

(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 \( \Phi \) of assertions. A cutting plane is a plane through the solution space of the real relaxation of \( \Phi \) that cuts off some of the non-integer-compatible assignments. More precisely, \( \sum c_i x_i = d \) is a cutting plane for \( \Phi \) 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 \( \Phi \) also satisfies \( \mathcal{H} \); and (ii) at least one assignment satisfying the real relaxation of \( \Phi \) also satisfies \( \neg \mathcal{H} \).

The inequality \( \mathcal{H} \) can be safely added to \( \Phi \) without changing any of the (integer-compatible) satisfying assignments. A cut is always entailed by the integer-tightening of \( \Phi \) and never by the real relaxation of \( \Phi \). 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.

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 balanced-solve shown in Figure 1. First, an efficient incremental exact solver exact-solve is called with a heuristic cap on the number of pivots it may perform, \( k_{EX} \). We assume that exact-solve returns a status \( c \) (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 \( s = \text{orig}(s) = \sum c_i x_i \) is added to \( T \) as \( \bar{s} = \sum \text{float}(c_i) \cdot \bar{x}_i \), where the conversion function float maps a rational to the nearest float.

Each variable \( \bar{x}_i \), the bounds \( l(x) \) and \( u(x) \) are constructed from the \( \delta \)-rationals, \( l(x) \) and \( u(x) \) by approximating \( \delta \) as a small constant \( \epsilon \). For example, if \( l(x) = (c, d) \), then \( \bar{l}(x) \) becomes 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 \( \bar{a} \) as well as the final set of basic variables \( \bar{B} \) from the LP solver. The assignment \( \bar{a} \) is converted into a rational assignment \( a' \) by the import-assignment routine (given below). The seed-exact procedure takes \( \bar{B} \) 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 \( k_{FP} \). For final calls to balanced-solve (i.e. the DPLL(T) SAT engine has found a propositionally satisfying assignment), \( k_{FP} \) 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.

\(^{3}\)Often, an additional requirement is that \( \mathcal{H} \) is not satisfied by the current assignment \( a \). We will not require this here.
with, but also partly because the discarded portion often corresponds exactly to a rounding error. For each variable \( x \) in the assignment, \textsc{ImportAssignment} first approximates \( \tilde{a}(x) \) 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 \( \epsilon \) of the last known assignment for \( x \) in the exact solver. If so, the last known assignment is used. Next, if \( x \in \mathcal{V}_2 \) and the value is within \( \epsilon \) of an integer \( z \) (\( \lfloor r \rfloor \) denotes the nearest integer to \( r \)), then \( z \) is used. Finally, \textsc{ImportAssignment} examines the value with respect to \( l(x) \) and \( \tilde{u}(x) \). If the value violates one of these bounds or is within \( \epsilon \) of a bound, then the bound is used instead.

The \textsc{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 \textsc{Update} on each non-basic variable. Next it computes the set, \( B' \), 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 \( B' \) 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 \( B' \) meaning it is non-basic but should be basic, and \( T_{i,j} \neq 0 \) and \( x_i \not\in \bar{B} \) 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 \( \tilde{a}'(x_i) \). Approximations made by the LP solver or by \textsc{ImportAssignment} mean that \textsc{SeedExact} may fail to detect a satisfying assignment or a conflict in which case it returns \text{Unknown}. The \textsc{SeedExact} procedure can be seen as achieving a similar effect as \textsc{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 \textit{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 \textsc{IntegerSolve} algorithm (Fig. 4) illustrates our approach. First, the real relaxation of the problem is solved using the \textsc{BalancedSolve} algorithm described above. If the real
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 unsatisifiable, then we are done. Otherwise, we construct a MIP instance and call an MIP solver (with a pivot limit \( k_{\text{MIP}} \)) to search for an integer-compatible solution. When \( \text{Unsat} \) is returned, we also retrieve a proof tree \( t \), 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 \( \text{Replay} \) procedure described below. Otherwise, if \( \text{Sat} \) is returned, we attempt to assign the verification as before. If the verification fails, we again call \( \text{ExactSolve} \) to ensure that we have a solution to the real relaxation before continuing. If we are unable to verify that the problem is \( \text{Unsat} \) 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 \( \forall v \in V \subseteq \mathbb{R} \). Let \( \bar{P} \) 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 \( \bar{P} \) is integer-infeasible can be described at an abstract level as a search tree. The root node represents the initial problem \( \bar{P} \) and each non-root node is derived from its parent by 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 \( \langle h, E \rangle \), where \( h \) is an inequality constraint and \( E \) is an explanation, a (possibly empty) finite set, each element of which is either some \( h' \) where \( \langle h', E' \rangle \) appears earlier in \( N \) or is a constraint from the initial problem \( \bar{P} \). We denote by \( C_N \) the set \( \{ h \mid \langle h, E \rangle \in N \} \).

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 \( \text{Propagate} \) 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 \( \text{Branch} \) 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 \( \bar{a} \) 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 \( \alpha \) and enforces the constraint \( v \leq [\alpha] \lor v \geq [\alpha] \). The rule has two children, each of which 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 \( \bar{P} \cup C_N \) 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 \( \text{Replay} \) function.

The \( \text{Replay} \) 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 \( \text{Replay} \) 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 \( \text{Replay} \) procedure returns \( \text{Unknown} \) if the replay has failed.)

If \( t \) is a leaf node, then \( P \cup C_N \) should be integer-infeasible. We check the exact analog, \( P \cup C_H \). If unsuccessful, we fail, returning \( \text{Unknown} \); otherwise, we return a conflict. To compute the conflict, we make use of an auxiliary function, \( \text{Regress} \), which is not shown. \( \text{Regress} \) 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 \( \text{Propagate} \) 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 \( \text{REPLAY} \). 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 \( \text{BRANCH} \) 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 \( \text{REPLAY} \) recursively on each of the two sub-trees, passing one of the branch conditions to each sub-tree. The \( \text{RESOLVE-BRANCH} \) 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 \( \text{REGRESS} \) to ultimately construct a conflict with constraints in \( P \) (we require \( \text{REGRESS} \) to return Unknown if \( K \) is Unknown).

Lines 12 and 13 of \( \text{REPLAY} \) require converting \( (\bar{h}, \bar{E}) \) to an exact analog, \( (h, E) \), 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 \( \text{Gomory cutting planes} \).

The MIP solver can add a Gomory cutting plane \( \bar{h} \) when the following conditions hold: (i) there is a row in \( \bar{T} \), \( b_i = \sum \bar{T}_{i,j} \cdot x_j \); (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 \( E \). For simplicity of presentation, we additionally assume all of the variables are integer and all the coefficients \( \bar{T}_{i,j} \) 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, \( \bar{a}(b_i) = \sum \bar{T}_{i,j} \cdot \bar{u}(x_j) \). The cut \( h \) for these constraints is then
\[
\sum \bar{T}_{i,j} \cdot \bar{u}(x_j) \geq 0.
\]

Given \((\bar{h}, \bar{E})\), we can attempt to derive a trusted cut and explanation \((h, E)\) as follows. To reconstruct the cut, for every bound \( x_j = \bar{u}(x_j) \in \bar{E} \), 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 \( b_i = \sum \bar{T}_{i,j} x_j \) in exact precision as a row vector \( \alpha \). The coefficient for the basic variable in \( \alpha \) is \(-1\) (\( \alpha_i = -1 \)). Non-basic variables’ coefficients are estimated from the approximate variables, \( \alpha_j = \text{DIOAPPROX}(T_{i,j}, D) \). If after approximation, the sign of \( \alpha_j \) does not match the sign of \( T_{i,j} \), this cut cannot be reproduced. The equalities \( TV = 0 \) entail \( \alpha_k x_k = 0 \) iff \( \alpha \) is in the row span of \( T \). This entailment can be checked by replacing auxiliary variables with their original definitions,
\[
\alpha_i \cdot x_i + \sum \alpha_j \cdot x_j + \sum \alpha_k \cdot \text{orig}(x_k),
\]
and rejecting this cut if any of the coefficients do not cancel to 0. The row \( \alpha \) 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 \( h \)
\[
h : \sum \frac{\alpha_j}{b - [b]} (u(x_j) - x_j) \geq 1
\]
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]. 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 \( \text{EXACTSOLVE} \) procedure with a pivot cap of \( k_{EX} = 200 \) in Fig. 1 (with \( k_{FL} = 200 \) for non-final calls). Values of other parameters used in our experiments are \( D = 2^{26}, \epsilon = 10^{-9}, k_{LP} = 10000; \) and \( k_{MIP} = 200000 \). 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. 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 \( \frac{a}{b} \) where \( \log_2(|a|) + \log_2(|b|) \geq 512 \).

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 \text{QF_LRA} family miplib (which are derived from benchmarks in [18]) in something closer to their original form.\(^7\)

\[^7\]\(\alpha\) can also be generated by Gaussian elimination from \( \bigwedge_k x_k = \text{orig}(x_k) \).

\[^8\]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 a358be).

\[^10\]A comparison of other solvers on the miplib problems after this reencoding gave similar results to those reported in Table 1.
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_LIA 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. 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 AlterEgo [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.)

11The conjunctive families are dillig, milp2003, 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.

TABLE I: Experimental results on QF_LRA and QF_LIA benchmarks.

<table>
<thead>
<tr>
<th>Sat</th>
<th>Unsat</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPsolve calls</td>
<td>attempts successes</td>
</tr>
<tr>
<td>CVC4+MIP</td>
<td>CVC4</td>
</tr>
<tr>
<td>CVC4+MIP</td>
<td>662</td>
</tr>
<tr>
<td>CVC4</td>
<td>-</td>
</tr>
<tr>
<td>yices2</td>
<td>-</td>
</tr>
<tr>
<td>mathsat5</td>
<td>-</td>
</tr>
<tr>
<td>Z3</td>
<td>-</td>
</tr>
<tr>
<td>altergo</td>
<td>-</td>
</tr>
<tr>
<td>cutsat</td>
<td>-</td>
</tr>
<tr>
<td>scip</td>
<td>-</td>
</tr>
<tr>
<td>glpk</td>
<td>-</td>
</tr>
</tbody>
</table>

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 MSIP was invoked at least once, and collected the following statistics: the number of times MSIP was called, the number of attempts and successes at verifying Sat results from MSIP, and the number of attempts and successes at replaying Unsat results from MSIP. The results are shown in Table II.

On QF_LIA 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 INTEGER-SOLVE 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. calypso 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 \( \epsilon \), where \( \epsilon \) 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