Simplex with Sum of Infeasibilities for SMT

Tim King* Clark Barrett* Bruno Dutertre†

*New York University †SRI International

***Abstract*—The de facto standard for state-of-the-art real and integer linear reasoning within Satisfiability Modulo Theories (SMT) solvers is the Simplex for DPLL(T) algorithm given by Dutertre and de Moura. This algorithm works by performing a sequence of local optimization operations. While the algorithm is generally efficient in practice, its local pivoting heuristics lead to slow convergence on some problems. More traditional Simplex algorithms minimize a global criterion to determine the feasibility of the input constraints. We present a novel Simplex-based decision procedure for use in SMT that minimizes the sum of infeasibilities of the constraints. Experimental results show that this new algorithm is comparable with or outperforms Simplex for DPLL(T) on a broad set of benchmarks.**

I. Introduction

The simplex algorithm introduced by Dutertre and de Moura in [1] for use in the DPLL(T) framework is the core reasoning module for linear arithmetic in nearly every state-of-the-art Satisfiability Modulo Theories (SMT) solver including CVC4, MathSAT, OpenSMT, SMTInterpol, Yices, and Z3 [2], [3], [4], [5], [6]. The algorithm—which we will call SimplexForSMT—relies on specific pivoting heuristics to search for a satisfying model or a conflict. Many pivot choices are possible and those choices can dramatically change the search for a solution. The heuristic pivot selection scheme that many SMT solvers use is based on local criteria and is potentially subject to cycling: it may return to the same basis state infinitely often. Solvers employ tactics to detect cycling, and slowly edge towards pivot-selection rules that guarantee termination, such as Bland's Rule [7], [8], [9]. Unfortunately, Bland's rule converges very slowly and is not effective on hard problems that require many pivots.

Before SimplexForSMT, earlier simplex-based approaches for SMT used repeated optimization (via an algorithm like Primal in Section III) as constraints arrived [15], [16], [17]. Since its initial publication, little work has been published on directly improving the simplex solver itself. Griggio's thesis [13] gives a number of details on implementation and additional pivoting heuristics. Most recent work on QF_LRA has focused on combining floating point and exact precision solvers [18], [19], [20].

In the more traditional setting, Simplex is used to minimize (or maximize) a linear function f. Throughout execution of the Simplex algorithm, the value of f never increases. As long as f strictly decreases, no cycling is possible. Thus, specialized techniques to prevent cycling are only required to break out of sequences of degenerate pivots, that is, pivots that do not change f. Procedures can then be designed around two different modes: a heuristic mode that is efficient in practice, and a mode for escaping degeneracy.

This paper proposes an adaptation for SMT of the sum-of-infeasibilities method from the Simplex literature [7], [8]. We call this method SOISimplex. Minimizing the sum-of-infeasibilities provides a witness function similar to f which accomplishes several things at once: it helps guide the search towards both models and conflicts; it prevents cycling; and it can be used to determine when to safely re-enable aggressive heuristics without losing termination.

In other aspects, SOISimplex is similar to the SimplexForSMT algorithm, providing similar features and having similar performance on many problems. However, its performance is noticeably better on certain problem instances that require many pivots.

The rest of the paper is organized as follows. Section II covers basic background material on SMT, DPLL(T), and linear real arithmetic. Section III describes a naive traditional primal simplex optimization routine. Section IV gives a description of SimplexForSMT. Section V then describes the new SOISimplex algorithm. Empirical results are given in Section VI, and Section VII concludes.

II. Background

The basic SMT problem is to determine whether a formula is satisfiable with respect to some fixed first-order theory T. Modern SMT solvers rely on an architecture called DPLL(T) which integrates a fast SAT solver with one or more theory solvers for specific first-order theories [10]. The SAT solver reasons about the Boolean skeleton of the formula, allowing the theory solvers to reason only about conjunctions of literals in their theory. This paper's main concern is a novel theory solver for quantifier-free linear real arithmetic (QF_LRA).

A formula in QF_LRA is a Boolean combination of atoms of the form $\sum c_j \cdot x_j \bowtie d$, where c_j, d are rational, $\bowtie \in \{=, \leq, \geq\}$, and $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ is a set of variables. By using simple transformations [1], the set of constraints presented to a QF_LRA theory solver 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 will refer to the entry in row i and column j of T as $t_{i,j}$, and use r_i to denote the i-th row of T. We further 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. If no such assignment

ISBN 978-0-9835678-3-7/13.

1: **procedure** UPDATE(j, δ)
2: $a(x_j) \leftarrow a(x_j) + \delta$
3: **for all** $i | t_{i,j} \neq 0$ **do**
4: $a(x_i) \leftarrow a(x_i) + t_{i,j} \cdot \delta$

(a) Changing $a(x_j)$ by δ ($x_j \in \mathcal{N}$)

1: **procedure** PIVOT(i, j)
2: $r_j \leftarrow r_j - \frac{1}{t_{i,j}} \cdot r_i$
3: **for all** $k | t_{k,j} \neq 0 \wedge k \neq j$ **do**
4: $r_k \leftarrow r_k + t_{k,j} \cdot r_j$
5: $\mathcal{B} \leftarrow (\mathcal{B} - \{b_i\}) \cup \{x_j\}$
6: $\mathcal{N} \leftarrow \mathcal{V} \setminus \mathcal{B}$

(b) Pivot b_i and x_j ($t_{i,j} \neq 0$)

1: **procedure** UPDATEANDPIVOT(j, δ, i)
2: UPDATE(j, δ)
3: **if** $i \neq j$ **then**
4: PIVOT(i, j)

(c) Composing Update and Pivot

Fig. 1: Algorithms for maintaining $Ta = 0$

exists, the theory solver must detect a *conflict* and provide an *explanation* (cf. [10], [1]), an infeasible subset (preferably small) of the set of constraints presented to the theory solver.

In all of the algorithms in this paper, 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 *nonbasic* 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 the i-th row of T is all zeroes iff $x_i \in \mathcal{N}$. Furthermore, for each column i such that $x_i \in \mathcal{B}$, we have $t_{k,i} = 0$ for all $k \neq i$ and $t_{i,i} = -1$. Thus, each nonzero row r_i of T represents a constraint $b_i = \sum_{x_j \in \mathcal{N}} t_{i,j} \cdot x_j$. It is sometimes convenient to use the matrix obtained by adding the identity matrix to T. We define $\tau = T + I$ and refer to the entry in row i and column j of τ as $\tau_{i,j}$. Note that on the diagonal, $\tau_{i,i} = 0$ for $b_i \in \mathcal{B}$ and $\tau_{j,j} = 1$ for $x_j \in \mathcal{N}$ (off the diagonal, $\tau_{i,j} = t_{i,j}$). The column length for a variable x_j, denoted by $|\operatorname{Col}(j)|$, is the number of nonzero entries in column j.

The algorithms in this paper work by making a series of changes to an initial assignment a until the constraints are satisfied or determined unsatisfiable. During this process, $Ta = 0$ is an invariant. To initially satisfy this invariant, one can set $a(x_i) = 0$ for all $x_i \in \mathcal{V}$. To maintain the invariant, whenever the assignment to a nonbasic variable changes, the assignments to all dependent basic variables are also updated (Fig. 1a). The main ingredient of Simplex-based procedures is the *pivoting operation* shown in Figure 1b. Pivoting takes a basic variable b_i and a nonbasic variable x_j such that $t_{i,j} \neq 0$, and swaps them: after pivoting, x_j becomes basic and b_i becomes nonbasic. Figure 1c gives the composition of the update and pivot operations, UPDATEANDPIVOT.

1: **procedure** PRIMAL(f)
2: **while** $\operatorname{Flex}(f) \neq \emptyset$ **do**
3: UPDATEANDPIVOT(PRIMALSELECT())
4: **return** $a(f)$

(a) PRIMAL(f) with a generic selection routine

1: **procedure** PRIMALSELECT
2: $S \leftarrow \emptyset$
3: **for all** $x_j \in \operatorname{Flex}(f)$ **do**
4: $S \leftarrow S \cup \langle j, k \rangle$, where $\langle | \delta_{\mathrm{B}}(j,k) |, k \rangle$ is minimal
5: select $\langle j, k \rangle \in S$ minimizing $\langle - | \operatorname{sgn}(\delta_{\mathrm{B}}(j,k)) t_{0,j} |, j \rangle$
6: **return** $\langle j, \delta_{\mathrm{B}}(j,k), k \rangle$

(b) PRIMALSELECT with a terminating variant of Dantzig's rule

Fig. 2: Primal Simplex

III. NAIVE PRIMAL SIMPLEX

The classic problem in linear optimization is to find an assignment a that satisfies the linear equalities $Ta = 0$ and the bounds $l \leq a \leq u$, and that minimizes a linear function $f = \sum_{x_k \in \mathcal{V}} c_k \cdot x_k$. The problem can be solved with the PRIMAL Simplex algorithm shown in Figure 2. It is typical to assume that the algorithm is given an initial feasible assignment as input, so that both $Ta = 0$ and $l \leq a \leq u$ are initially satisfied.

The optimization function f is treated as a special additional variable $f = x_0 = \sum_{x_k \in \mathcal{V}} c_k \cdot x_k$. We add a row and column to T (for convenience, at the top and left, indexed by 0), with $t_{0,j} = c_j$, for $1 \leq j \leq n$, $t_{i,0} = 0, 1 \leq i \leq n$, and $t_{0,0} = -1$. The entries in the new row corresponding to basic columns can be set to zero using matrix row additions (as is done in PIVOT). We can then treat f (which we use as another name for x_0 below) as a basic variable with no bounds. (Note that to instead maximize f with the same machinery, we simply minimize its negation $-f$.)

Every round of PRIMAL begins by checking whether or not f is currently at its minimum. This is done by looking at the assignments to each nonbasic variable on f's row. The value of x_j that minimizes f–call this v_j–is $u(x_j)$ if $t_{0,j}$ is negative and $l(x_j)$ if $t_{0,j}$ is positive (ignoring other constraints). If $a(x_j) = v_j$ for each nonbasic variable x_j on f's row (where $t_{0,j} \neq 0$), then the current value of f, $a(f)$, must be the minimum because we can prove $f \geq a(f)$ as follows:

$$\begin{aligned} f &= \sum_{\tau_{0,j}>0} t_{0,j} x_j + \sum_{\tau_{0,k}<0} t_{0,k} x_k \\ &\geq \sum_{\tau_{0,j}>0} t_{0,j} l(x_j) + \sum_{\tau_{0,k}} t_{0,k} u(x_k) \\ &= \sum_{\tau_{0,j}>0} t_{0,j} a(x_j) + \sum_{\tau_{0,k}<0} t_{0,k} a(x_k) = a(f) \end{aligned} \tag{1}$$

The search can then terminate. Otherwise, there is some x_j on f's row s.t. $a(x_j) \neq v_j$, and it is unclear whether $a(f)$ is at a minimum. By trying to change $a(x_j)$ for these x_j, we can at the same time hunt for an assignment that decreases $a(f)$

and search for a proof of optimality. We will call the non-basic variables on f's row whose assignments are not at their relevant bounds the *flexible* variables for this row. The set of flexible variables for an arbitrary basic variable b_i is denoted $\mathrm{Flex}(d, b_i)$ where d is a *directional* rational that is used as an implicit multiplier:

$$\begin{aligned}\mathrm{Flex}(d, b_i) = &\{x_j | d \cdot \tau_{i,j} > 0 \wedge a(x_j) > l(x_j)\} \cup \\ &\{x_k | d \cdot \tau_{i,k} < 0 \wedge a(x_k) < u(x_k)\}\end{aligned} \tag{2}$$

The parameter d allows us to choose whether to minimize or maximize b_i and will be discussed further in Sections IV and V. When $d = 1$ (as it always is in this Section), we will drop the first argument to Flex as a notational convenience. Thus, f is at its minimum when $\mathrm{Flex}(f) = \emptyset$.

To decrease the value of $a(f)$, we choose some $x_j \in \mathrm{Flex}(f)$ and determine an appropriate δ for $\textsc{Update}(x_j, \delta)$ (we discuss the strategy for picking x_j below). The direction in which we attempt to move $a(x_j)$ is determined by $t_{0,j}$: if $t_{0,j} < 0$, then we want $\delta \geq 0$ and if $t_{0,j} > 0$, then we want $\delta \leq 0$. Since the UPDATE operation must maintain the invariant $l \leq a \leq u$, the value of δ is constrained by the bounds on x_j: $l(x_j) \leq a(x_j) + \delta \leq u(x_j)$. Also, for every b_i that depends on x_j, the value $a(b_i)$ must stay within bounds: $l(b_i) \leq a(b_i) + t_{i,j} \cdot \delta \leq u(b_i)$. These cases can be unified using τ: for all k, $l(x_k) \leq a(x_k) + \tau_{k,j} \cdot \delta \leq u(x_k)$.

PRIMAL always considers $\textsc{Update}(j, \delta)$ operations that are maximal: the value of δ is selected so that at least one variable's assignment is pushed against its bound (any larger change would violate the bound). For each k, the candidate value for δ is the one that sets x_k equal to one of its bounds (which bound is determined by the sign of δ and the sign of $\tau_{k,j}$). We call these candidate values for δ the *break points* of x_j. Formally, let $\delta_{\mathrm{U}}(j, k, \alpha)$ be the amount x_j must change in order to make x_k equal to α after an UPDATE:

$$\delta_{\mathrm{U}}(j, k, \alpha) = \frac{1}{\tau_{k,j}} (\alpha - a(x_k)), \text{and}$$

$$\delta_{\mathrm{B}}(j, k) = \begin{cases} \delta_{\mathrm{U}}(j, k, l(x_k)) & t_{0,j} \cdot \tau_{k,j} > 0 \\ \delta_{\mathrm{U}}(j, k, u(x_k)) & t_{0,j} \cdot \tau_{k,j} < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The break points for x_j are all defined values of $\delta_{\mathrm{B}}(j, k)$.

In PRIMAL, for each j, we simply select k to minimize $|\delta_{\mathrm{B}}(j, k)|$ (ties can be broken by picking the minimum k). The operation $\textsc{Update}(j, \delta_{\mathrm{B}}(j, k))$ then maintains the invariant that no variable violates its bound. Additionally, the assignment to x_k is guaranteed to be pressed up against its bound. When $j \neq k$, x_k is a basic variable, so we can allow for (potential) future progress by pivoting x_k out of the basis and replacing it with x_j. The operation $\textsc{UpdateAndPivot}(j, \delta_{\mathrm{B}}(j, k), k)$ then maintains both the invariant $Ta = 0$ and $l \leq a \leq u$.[1] Because x_k leaves the basis, our strategy of minimizing $|\delta_{\mathrm{B}}(j, k)|$ to select k is called a *leaving* rule. By always selecting updates like this, PRIMAL ensures that $a(f)$ monotonically decreases.

We have just described a rule for selecting x_k given x_j, but we need an *entering* rule for selecting x_j. A simple way to ensure termination is to select the entering variable x_j with the smallest index j. This style of selecting entering and leaving variables is called Bland's rule in the literature, and its termination is a classic result of linear programming [9], [8], [7]. A better heuristic is to select x_j so as to maximize the value of $|t_{0,j}|$. This is called Dantzig's rule.[2]

The algorithm $\textsc{Primal}(f)$ in Figure 2 is a minimization routine that repeatedly selects an update and pivot until $\mathrm{Flex}(f)$ is empty and then returns the minimum value found for f.[3] The selection procedure uses a terminating variant of Dantzig's rule (it follows Dantzig's rule as long as $\delta_{\mathrm{B}}(j, k)$ is nonzero, otherwise it follow's Bland's rule). Note that when $\delta_{\mathrm{B}}(j, k) \neq 0$, the value of f strictly decreases, which makes it impossible to return to any previous state (as all previous states had larger values of f). Thus, the presense of a minimization function makes it easier to rule out cycles (the source of nonterminating runs). Termination only needs to be addressed for cases when f gets stuck and stops decreasing.

IV. SIMPLEX FOR DPLL(T)

The SIMPLEXFORSMT algorithm from [1] is tightly tuned to the DPLL(T) framework. It is designed to support incremental processing of arithmetic literals and efficient backtracking, and it computes minimal explanations in case of conflicts. Strict inequalities are encoded using an implicit infinitesimal variable δ (see [1] for details on δ-rationals).

In the DPLL(T) framework, a SAT solver incrementally sends theory literals to the theory solver. Periodically, it queries the solver about the current set of literals, expecting that the solver will either report satisfiable (with a satisfying assignment) or unsatisfiable (with a conflict). With appropriate preprocessing, we can assume that the linear equalities $Ta = 0$ are fixed (modulo pivoting) from the beginning, that all variables are initially unbounded, and that the theory literals sent by the SAT solver are of the form $x_i \leq c$ or $x_i \geq c$. The literals sent thus determine the bound constraints: $l \leq \mathcal{V} \leq u$. As in PRIMAL, the invariant $Ta = 0$ is always maintained. This is done by starting with $a(x) = 0$ for all x and by using only UPDATE to change variable assignments. The main job of SIMPLEXFORSMT then is to modify the current assignment using UPDATE until it satisfies the bounds or report a conflict if this is impossible. This is done by the SIMPLEXFORSMTCHECK routine shown in Figure 3.

This routine focuses on searching for an assignment a that satisfies $l \leq a \leq u$. We say that x is an *error variable* if a violates one of the bounds on x, and we denote by E the set

[1] When $k = j$, $\textsc{UpdateAndPivot}(j, \delta_{\mathrm{B}}(j, j), j)$ corresponds to an update without a pivot.

[2] Dantzig's rule tends to be dominated in practice by more sophisticated rules such as steepest gradient descent [9], [8], [7].

[3] For the purposes of this paper, we have ignored unbounded problems, i.e. problems where $a(f)$ can take on arbitrarily low values [9], [8], [7]. To handle this case, change the while loop condition additionally to stop once $a(f)$ is set to $-\infty$.

of error variables. Let $\mathrm{Vio}(x)$ denote the amount by which x violates its bound:

$$\mathrm{Vio}(x) = \begin{cases} l(x) - a(x) & a(x) < l(x) \\ a(x) - u(x) & a(x) > u(x) \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

Thus, $\mathrm{Vio}(x)$ is nonnegative and piecewise linear, and x satisfies its bounds iff $\mathrm{Vio}(x) = 0$. Finding a satisfying assignment requires reducing each $\mathrm{Vio}(x_i)$ to 0. Locally, minimizing a $\mathrm{Vio}(x_i)$ is equivalent to minimizing $d_i \cdot x_i$ where d_i is 1 if $a(x) > u(x)$, -1 if $a(x) < l(x)$, and 0 otherwise.

In Fig. 3, the first loop ensures that the nonbasic variables satisfy their bounds (lines 3-4). The main work of the routine is the second loop which focuses on finding updates to basic variables that are in E. When $\mathrm{E} = \emptyset$, the current assignment is feasible and the search stops. Otherwise, there is some $b_i \in \mathrm{E}$.

The set $\mathrm{Flex}(d_i, b_i)$ contains the nonbasic flexible variables of row i that enable the function $d_i \cdot b_i$ to decrease. If $\mathrm{Flex}(d_i, b_i)$ is nonempty, then a variable $x_j \in \mathrm{Flex}(d_i, b_i)$ is chosen; b_i is pivoted with x_j; and the assignment to x_j is updated enough to move b_i to its violated bound. Let $\mathrm{VB}(b_i)$ denote the violated bound on b_i (either $l(b_i)$ or $u(b_i)$). Then for $x_j \in \mathrm{Flex}(d_i, b_i)$, the operation $\textsc{Update}(j, \delta_{\mathrm{U}}(j, i, \mathrm{VB}(b_i)))$ will set $a(b_i)$ to the violated bound.

If $\mathrm{Flex}(d_i, b_i)$ is empty, then the bounds on the nonbasic variables on b_i's row imply that $d_i \cdot b_i$ is at a minimum value so there is no way to satisfy $d_i \cdot b_i \leq d_i \cdot \mathrm{VB}(b_i)$ without violating some other bound. Thus, the current set of bounds is unsatisfiable. We can compute an explanation by collecting all of the contributing bounds on row i:

$$\bigwedge_{d_i \cdot \tau_{i,j} > 0} x_j \geq l(x_j) \wedge \bigwedge_{d_i \cdot \tau_{i,k} < 0} x_k \leq u(x_k) \wedge d_i \cdot b_i \leq d_i \cdot \mathrm{VB}(b_i)$$

We denote the conflict explanation generated in this fashion as $\mathrm{RC}(i)$. This explanation is minimal (if any constraint is removed, the remaining constraints are satisfiable) [11]. Upon detection, the row conflicts are added to the set of conflicts $\mathcal{C}$.

To ensure termination, it is sufficient to always select the minimum $b_i \in \mathrm{E}$ to leave the basis, and the minimum $x_j \in \mathrm{Flex}(d_i, b_i)$ (a variation of Bland's rule). However, the dominant heuristic in state-of-the-art implementations of SIMPLEXFORSMT is to instead select the $x_j \in \mathrm{Flex}(d_i, b_i)$ with minimum column length $|\mathrm{Col}(j)|$. This heuristic works quite well in practice but is not guaranteed to terminate. (The function $\mathrm{Vio}(b_i)$ will decrease to 0 for b_i but it may increase for other basic variables or for x_j.) A simple means of ensuring termination is to count the number of pivots and switch to Bland's rule once this passes a finite cap, This strategy is shown in Figure 3. The variable pc is the pivot count and, once pc reaches some threshold H, the pivot selection heuristic switches to Bland's rule. This strategy or slight variations of it are currently used by default in CVC4, MathSat [13], OpenSMT [14], Yices, Yices 2, and Z3 [12].

An improvement to the algorithm (and a contribution of this paper) can be obtained by implementing a more aggressive conflict detection. Instead of only checking the row of the first basic variable in error for a conflict, all rows are checked for conflicts. This variation (a replacement for the CHECKFORCONFLICT(I) procedure in Fig. 3 is shown in Figure 4. To implement this efficiently, we keep track of the size of $\mathrm{Flex}(\pm 1, b_i)$ for all $b_i \in \mathcal{B}$. These counts depend on the coefficients $t_{i,j}$, and the relationships $a(x_j) < u(x_j)$ and $a(x_j) > l(x_j)$. The bookkeeping for keeping these counts accurate is amortized into the theory solver operations. Conflict detection then amounts to checking when $|\mathrm{Flex}(d_i, b_i)|$ is 0 for $b_i \in \mathrm{E}$. (Only b_i that were affected in the previous iteration need to be tested for conflicts.) An evaluation of SIMPLEXFORSMT with and without this optimization (using CVC4) showed a 46% speedup on the `QF_LRA` SMT-LIB benchmarks. This optimization is on by default in CVC4.

```
1: procedure SIMPLEXFORSMTCHECK
2:   pc ← 0
3:   while ∃ x_j ∈ 𝒩 ∩ E do
4:     UPDATE(x_j, −d_j · Vio(x_j))
5:   while E ≠ ∅ ∧ 𝒞 = ∅ do
6:     UPDATEANDPIVOT(SIMPLEXFORSMTSELECT())
7:     pc ← pc +1
8:   return 𝒞 = ∅ ? Sat(a) : Unsat(𝒞)
9: procedure SIMPLEXFORSMTSELECT
10:  select b_i from E to minimize i
11:  CHECKFORCONFLICT(i)
12:  if 𝒞 ≠ ∅ then
13:    return ⟨i, 0, i⟩
14:  h ← pc < H ? 1 : 0
15:  select x_j from Flex(d_i, b_i) to minimize ⟨h·|Col(j)|, j⟩
16:  return ⟨j, δ_U(j, i, VB(i)), i⟩
17: procedure CHECKFORCONFLICT(i)
18:  if Flex(d_i, b_i) = ∅ then
19:    𝒞 ← {RC(i)}
```

Fig. 3: Check procedure for SIMPLEXFORSMT; uses a terminating selection rule and a procedure for detecting conflicts on row i

```
1: procedure CHECKALLCONFLICTS
2:   for all i|1 ≤ i ≤ n do
3:     if conflict on row i then
4:       𝒞 ← 𝒞 ∪ {RC(i)}
```

Fig. 4: Procedure that checks for all conflicts

V. SUM OF INFEASIBILITIES SIMPLEX

In this section, we introduce a Simplex-based theory solver for `QF_LRA` which we call SOISIMPLEX. Like SIMPLEXFORSMT in the previous section, it is designed to search for both conflicts and satisfying assignments in the context of a DPLL(T) search. It attempts to address the troubling lack of a straightforward global criterion for progress in SIMPLEXFORSMT by introducing a function to minimize. The function minimized is the sum of infeasibilities of all of the variables.

```
1: procedure SOICheck
2:   while ∃ x_j ∈ 𝒩 ∩ E do
3:     Update(x_j, −d_j · Vio(x_j))
4:   while Flex(f) ≠ ∅ ∧ 𝒞 = ∅ do
5:     UpdateAndPivot(SOISelect())
6:   if 𝒞 ≠ ∅ then
7:     return Unsat(𝒞)
8:   else if E = ∅ then
9:     return Sat(a)
10:  else
11:    return Unsat(SoiQE) (Sec. V-B)
12: procedure SOISelect
13:   CheckAllConflicts()
14:   if 𝒞 ≠ ∅ then
15:     return ⟨1, 0, 1⟩
16:   S ← ∅
17:   for x_j ∈ Flex(f) do
18:     L ← ∅
19:     for all k|k = j ∨ t_{k,j} ≠ 0 do
20:       L ← L ∪ {⟨δ_U(j, k, l(x_k)), k⟩}
21:       L ← L ∪ {⟨δ_U(j, k, u(x_k)), k⟩}
22:     select ⟨δ, k⟩ ∈ L to minimize ⟨ΔVio(j, δ), |δ|, k⟩
23:     S ← S ∪ ⟨j, δ, k⟩
24:   select ⟨j,δ,k⟩ ∈ S minimizing ⟨sgn(ΔVio(j, δ)) · |t_{0,j}|, j⟩
25:   return ⟨j, δ, k⟩
```

Fig. 5: SOICheck and selection rules for SOISimplex

For a given assignment, the sum of infeasibilities is given by: $\mathrm{Vio}(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}(x)$. Let Vio_F be the result of replacing $a(x)$ by x in the definition of Vio. The optimization function can be written as: $\mathrm{Vio}_F(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}_F(x)$. Minimizing the sum of infeasibilities is a standard technique for finding an initially feasible assignment for linear programs [7], [8].

We assume the same setup as in the previous section: we start with a fixed (modulo pivoting) tableau and a satisfying assignment a, and then the SAT solver sends a set of literals that determine the upper and lower bounds for the variables. The theory solver must provide a check routine that either reports satisfiable (with a satisfying assignment) or unsatisfiable (with a conflict). The main loop for SOISimplex uses essentially the same machinery to minimize $\mathrm{Vio}_F(\mathcal{V})$ as was used in Primal for minimizing a linear function f. However, there are a number of complications caused by the fact that $\mathrm{Vio}_F(\mathcal{V})$ is only piecewise linear instead of linear. The majority of this section is devoted to handling these challenges.

Because we cannot represent the optimization function $\mathrm{Vio}_F(\mathcal{V})$ directly in the tableau, we use a linearized approximation. First note that that $\mathrm{Vio}(\mathcal{V}) = \sum_{x \in \mathcal{V}} \mathrm{Vio}(x) = \sum_{x_i \in \mathcal{V}} d_i \cdot (a(x_i) - \mathrm{VB}(x_i))$. In some neighborhood of $a(x_i)$, the value of $d_i \cdot \mathrm{VB}(x_i)$ will be constant. Discarding this term and replacing $a(x_i)$ with x_i results in the function $f(\mathcal{V}) = \sum_{x_i \in \mathcal{V}} d_i \cdot x_i$. Note that the function still depends on the current assignment (which determines d_i), but for a given assignment, the function is linear. We can substitute for the basic variables and rearrange the sums to get:

$$f = \sum_{x_j \in \mathcal{N}} \left(\sum_{x_i \in \mathcal{V}} d_i \tau_{i,j} \right) \cdot x_j.$$

We use this function in roughly the same way we used f in Primal: it is the 0th variable and it is always basic. To compute the tableau row for f, we simply compute coefficients for each nonbasic variable x_j by adding, for each row i, the entry in column j multiplied by the directional multiplier d_i. The computed coefficients depend on d_i and thus have to be updated every time the assignment changes. This can be implemented efficiently by instrumenting Update to detect when d_i changes to d_i' for some i. When this happens, we update f's row (r_0) as follows: $r_0 \leftarrow r_0 + (d_i' - d_i) \cdot \tau_i$ (where τ_i is the i-th row of τ).

The check procedure for SOISimplex is given in Fig. 5. It iterates while: no row contains a conflict ($\mathcal{C} = \emptyset$), and there is a nonbasic variable on f's row with slack ($\mathrm{Flex}(f) \neq \emptyset$). If $\mathcal{C} \neq \emptyset$, then SimplexForSMTCheck safely terminates with the discovered conflict. If $\mathrm{Flex}(f)$ and E are empty, the current assignment is satisfying. Otherwise, $\mathrm{E} \neq \emptyset$, $\mathrm{Flex}(f) = \emptyset$, and f is at a minimum. Section V-B discusses extracting a conflict explanation with the SoiQE procedure.

As in the Primal algorithm, the selection procedure iterates over all $x_j \in \mathrm{Flex}(f)$. The leaving rule considers x_j as well as every basic variable b_k where $t_{k,j}$ is nonzero. We consider two possible updates (break points) for each such variable: one which sets it to its upper bound and one which sets it to its lower bound. Unlike Primal, we consider updates for which some new basic variable could become violated. However, we still ensure that global progress is made. We denote by $\Delta\mathrm{Vio}(j, \delta)$ the amount that $\mathrm{Vio}(\mathcal{V})$ would change if we were to change the current assignment by executing $\mathrm{Update}(j, \delta)$. From all of the possible leaving variables and updates, we then select the pair for which $\Delta\mathrm{Vio}(j, \delta)$ is minimal (equivalently, the pair that reduces the value of $\mathrm{Vio}(\mathcal{V})$ the most). Section V-A describes how to efficiently compute the values for $\Delta\mathrm{Vio}(j, \delta)$. We also show in that section that for each x_j, there is always a choice of $\langle \delta, k \rangle$ such that $\Delta\mathrm{Vio}(j, \delta) \leq 0$. This ensures that $\mathrm{Vio}(\mathcal{V})$ monotonically decreases. Tie breaking for the leaving rule is done by selecting the minimum value of $|\delta|$ and then the minimum variable index k. The motivation for the former is discussed in subsection V-C.

The entering rule selects between candidate triples $\langle j, \delta, k \rangle$ for $x_j \in \mathrm{Flex}(f)$. Any triple for which $\Delta\mathrm{Vio}(j, \delta)$ is negative ensures that SOISimplex is making progress. This allows for SOISimplex to treat $\mathrm{Vio}(\mathcal{V})$ in a manner analogous to $a(f)$ in Primal. Following our modified Dantzig's rule, we select the entering variable with the largest coefficient so long as it decreases $\mathrm{Vio}(\mathcal{V})$ with ties being broken by selecting the variable with the smaller index.

We show how SOISimplex works using the simple example shown in Fig. 6. With the given assignment, the bound $x_1 \geq 3$ is violated, and $\mathrm{Vio}(\mathcal{V}) = 2$. The variable x_2 is

$$
\begin{array}{rrcl}
T: & f & = -2 \cdot x_2 + x_3 & \\
 & x_1 & = 2 \cdot x_2 - x_3 & \\
3 \leq & x_1 & \leq 7 & a(x_1) = 1 \\
 & x_2 & \leq 3 & a(x_2) = 1 \\
1 \leq & x_3 & & a(x_3) = 1
\end{array}
$$

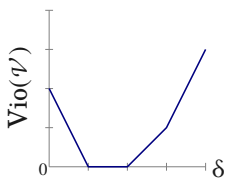

Fig. 6: Simple example showing $\mathrm{Vio}(\mathcal{V})$ after UPDATE(x_2, δ)

flexible, and we examine it for updates. The break points for x_2 are at $\delta \in \{1, 2, 3\}$, and correspond to changes to x_2 that respectively set x_1 to its lower bound, x_2 to its upper bound, and x_1 to its upper bound. Figure 6 shows how the value of $\mathrm{Vio}(\mathcal{V})$ changes if x_2 is updated by δ. For $\delta \in \{1, 2\}$, $\Delta\mathrm{Vio}(2, \delta) = -2$ and $\mathrm{Vio}(\mathcal{V})$ will become 0. Because of the tie-break on $|\delta|$, the pair $\langle \delta.k \rangle = \langle 1, 1 \rangle$ is selected, and then the triple $\langle 2, 1, 1 \rangle$ is returned. After the call to UPDATE, the algorithm terminates with a satisfying solution.

A. Computing $\Delta\mathrm{Vio}(j, \delta)$

To implement line 22 of SOISELECT, we must compute the values of $\Delta\mathrm{Vio}(j, \delta)$ for every break point δ. We use the fact that the function Vio_F is linear between break points and that the slopes of these linear segments can be computed. Let Δ be a increasing sorted list of the positive δ values in L, and let $\delta_0 = 0$: $0 = \delta_0 < \delta_1 < \ldots$. Let κ_i be the set of values of k that are paired with δ_i in L. We proceed as follows. We know that $\Delta\mathrm{Vio}(j, 0) = 0$ and that the slope β_0 as δ increases from 0 is $t_{0,j}$. Now, we can compute:

$$\Delta\mathrm{Vio}(j, \delta_i) = \Delta\mathrm{Vio}(j, \delta_{i-1}) + \beta_{i-1} \cdot (\delta_i - \delta_{i-1}).$$

Furthermore, we know that at δ_i, each variable x_k (for $k \in \kappa_i$) transitions to satisfying its bound or violating its bound, meaning that d_k will change at δ_i to some d'_k. This change can be used to compute the slope β_i for the next segment: $\beta_i = \beta_{i-1} + \sum_{k \in \kappa_i} (d'_k - d_k) \cdot \tau_{k,j}$. Continuing this walk over increasing values of δ computes $\Delta\mathrm{Vio}(j, \delta)$ for all $\delta \geq 0$. Another analogous pass can be done to compute the $\Delta\mathrm{Vio}(j, \delta)$ values for negative δ values. A number of nice properties follow from the above computation, including the the following lemma:

Lemma 1. *For each $x_j \in \mathrm{Flex}(f)$, there is some pair $\langle \delta, k \rangle \in$ L such that $\Delta\mathrm{Vio}(j, \delta) \leq 0$.*

Proof. If $\delta = 0$ is a break point, then $\Delta\mathrm{Vio}(j, 0) \leq 0$. Now assume 0 is not a break point. The x_j's considered are on f's row so $t_{0,j} \neq 0$. If $t_{0,j} > 0$, there must exist some $d_i \cdot \tau_{i,j} > 0$. So there exists a negatively-valued break point, $\delta_{\mathrm{U}}(j, i, \mathrm{VB}(i))$. Let δ be the negative break point closest to 0. We know that $\Delta\mathrm{Vio}(j, \delta) = 0 + t_{0,j} \cdot \delta < 0$. Similarly, if $t_{0,j} < 0$, then $\Delta\mathrm{Vio}(j, \delta) < 0$ for the minimal positive δ. □

The proof further suggests that it is sufficient to consider either just the negative or just the positive values of δ (depending on the value of $t_{0,j}$) without affecting correctness.

B. Conflicts with Multiple Rows

If $\mathrm{Flex}(f) = \emptyset$, $\mathcal{C} = \emptyset$ (i.e. no single row produces a conflict) but $\mathrm{E} \neq \emptyset$, we can still detect a conflict and derive an explanation as follows. Similar reasoning to that in (1) can be used to show that the sum of the assignments for the variables in E is strictly greater than the sum of their violated bounds:

$$\sum_{b_i \in \mathrm{E}} d_i \cdot a(b_i) > \sum_{b_i \in \mathrm{E}} d_i \cdot \mathrm{VB}(i).$$

So the bounds on the nonbasic variables in f's row and the basic variables in error cannot together be satisfied. This allows us to extract the following conflict explanation:

$$\bigwedge_{\tau_{0,j} > 0} x_j \geq l(x_j) \wedge \bigwedge_{\tau_{0,k} < 0} x_k \leq u(x_k) \wedge \bigwedge_{b_i \in \mathrm{E}} d_i b_i \leq d_i \, \mathrm{VB}(i)$$

Explanations constructed like this may not be minimal. However, we observe that for any subset S of E, if we construct the function $f_S = \sum_{x_j \in \mathcal{N}} \left(\sum_{b_i \in S} d_i \cdot t_{i,j} \right) \cdot x_j$, and $\mathrm{Flex}(f_S) = \emptyset$, then we can extract a smaller explanation using only the rows corresponding to basic variables in S. We use a number of heuristics and a straightforward adaptation of the QuickXplain algorithm [21] to attempt to find a minimal subset S that still generates a conflict (without additional Simplex search). Most of the time a conflict can be found with $|S| = 2$. In this case, the explanation is guaranteed to be minimal.

C. Termination

The termination of SOISIMPLEX is again based on the termination of Bland's rule. Suppose that SOISIMPLEX does not terminate. There are only a finite number of possible assignments that can be considered as the number of variables is finite, and every change to the assignment assigns a variable x_j to either $u(x_j)$ or $l(x_j)$. Because the value of $\mathrm{Vio}(\mathcal{V})$ is determined by the assignment and monotonically decreases, any nonterminating execution must have an infinite tail during which $\mathrm{Vio}(\mathcal{V})$ is unchanged and the update selected, $\langle j, \delta, k \rangle$ is such that $\Delta\mathrm{Vio}(j, \delta) = 0$. As was shown in the proof of Lemma 1, if the minimal $\Delta\mathrm{Vio}(j, \delta)$ found is 0, then $\delta = 0$ must be a break point. The leaving rule enforces that the δ selected minimizes the tuple $\langle \Delta\mathrm{Vio}(j, \delta), |\delta|, k \rangle$. So in the tail of a nonterminating execution $\Delta\mathrm{Vio}(j, \delta) = 0$ and $\delta = 0$ at every step. Thus after this point, no variable is changing in assignment and no variable changes its relationship to its bounds. Every leaving and entering variable is then selected based on picking the minimum index. The argument that PRIMAL cannot cycle under Bland's rule can then be directly applied. We refer readers interested in the proof of the termination of Bland's rule to [7], [8], [9].

D. Heuristics and $\mathrm{Vio}(\mathcal{V})$

Instead of examining all $x_j \in \mathrm{Flex}(f)$ for the best candidate, we can instead just look at heuristically many candidates. The search can stop once a candidate has been found that makes progress (i.e. $\Delta\mathrm{Vio}(j, \delta) < 0$). Further, there is more freedom in selection heuristics than we have shown here. In particular, one can use any heuristic desired until no

progress has been made for a while. CVC4's implementation for example uses a heuristic that prefers shorter columns until progress stalls and then uses Bland's rule.

During the calculation of break points, it is possible to determine if pivoting x_j with b_i would result in a row conflict on x_j's new row in $O(1)$ time by using the $|\operatorname{Flex}(\pm 1, b_i)|$ values. Such selections are always prefered. CVC4's selection also heuristically prefers the set E to be as small as possible.

VI. Experimental Results

In this section we describe two experiments. In the first, we compare CVC4 against itself using two different sets of options.[4] The first set of options uses the default solver, an implementation of SimplexForSMT (which is a bit better than the version that won the `QF_UFLRA` division—which includes `QF_LRA`—of SMT-COMP 2012 [22]). The second set of options enables a new implementation of SOISimplex. The two configurations of CVC4 are run with most other heuristics disabled so that the comparison is an accurate reflection of the performance of the two algorithms as described in this paper.[5] The comparison is done on the `QF_LRA` benchmarks from the `SMT-LIB` library [23] as well as a new family of benchmarks from biological modeling, `latendresse` [24]. The `latendresse` family of benchmarks is a set of problems that originated from an analysis of biochemical reactions using the flux-balance analysis method.[6] The `miplib` and `latendresse` families are of particular interest as they contain the only timeouts in these experiments. These problems are characterized by relatively little propositional structure, and a large and relatively dense input tableau. All of the experiments were conducted on a 2.66GHz Core2 Quad running Debian 7.0 with a time limit of 1000 seconds. Every example stays below a memory limit of 2GB. Overall, SOISimplex solves 636 while SimplexForSMT solves only 629. Interestingly, SOISimplex is slightly slower on the `SMT-LIB` benchmarks (see Fig. 7), and even solves one fewer benchmark (the satisfiable `miplib` benchmark `fixnet-7000.smt2`), but solves all of the `latendresse` benchmarks while SimplexForSMT times out on 8 of them.

To understand these results better, we recorded how many pivots were done (for both algorithms) during each call to the respective check routines (for benchmarks that both algorithms are able to solve). For the `SMT-LIB` benchmarks, almost all queries sent to the theory solver are "easy" for the simplex solvers (both SimplexForSMT and SOISimplex). Table I shows, for given numbers of pivots (or ranges of numbers of pivots), the number of calls to check whose pivot count is in that range. The maximum number of pivots for any single call to check is 2238. The number of pivots

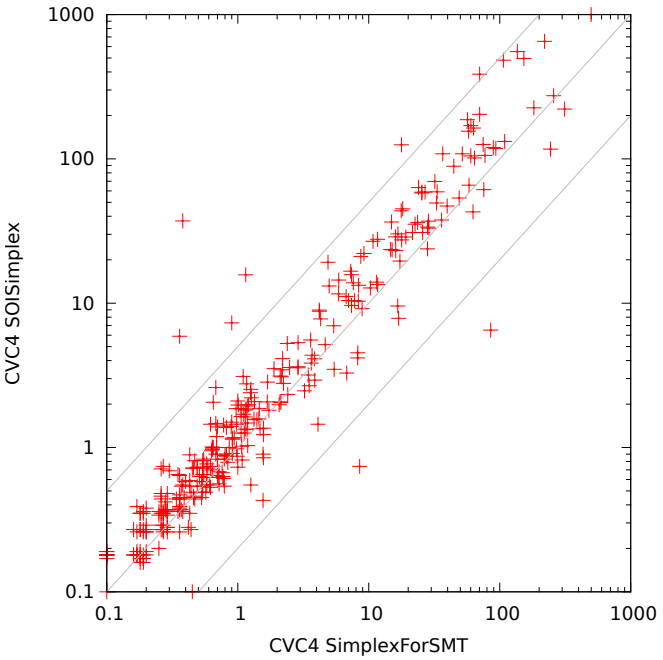

Fig. 7: Log-scaled running times (sec.) for experiment 1 on the `QF_LRA` benchmarks from `SMT-LIB`.

is generally very low and on average, SOISimplex uses fewer pivots than SimplexForSMT. The 8 timeouts by SimplexForSMT on `latendresse` have a very different signature. Each of them times out in the middle of a very long SimplexForSMTCheck call performing thousands of pivots. On average, the interrupted SimplexForSMTCheck routines had performed 18263 pivots and had been running 937s [/1000s]. This first experiment confirms our expectation that SOISimplex is effective at reducing the number of pivots required to solve a problem.

For the second experiment, we compare the same two algorithms in CVC4 against a number of state-of-the-art `QF_LRA` solvers. For this experiment, we enable a number of additional CVC4 options (those used in the SMT-EVAL 2013 run script) which are beyond the scope of this paper and which significantly improve performance (for both algorithms) on the `miplib` and `latendresse` benchmarks. These are disabled in the first experiment to better understand the relative strengths of the two algorithms on their own. The other solvers we compare with are: Z3 4.1.2 [4], mathsat 5.2.3 [3], yices 2.1.1, and OpenSMT 1.0.1[6]. Table II contains a summary of the number of problems solved by each solver and the cumulative time taken on the solved instances for the three families of benchmarks: all `SMT-LIB QF_LRA` benchmarks, the `miplib` family from `QF_LRA`, and the `latendresse` benchmarks.[7] The second experiment shows that the strongest overall solving strategy is obtained by using SOISimplex.

[4]Experiments were run using the submission to SMT-EVAL 2013: CVC4 version 1.2, available at github.com/CVC4/CVC4/tree/smteval2013.

[5]Both solvers are run with `--new-prop --no-restrict-pivots`. SOISimplex is run with the additional flag `--use-soi`. The `--no-restrict-pivots` flag disables stopping simplex after K pivots at non-leaf SimplexForSMTCheck calls ($K = 200$ by default).

[6]These benchmarks are available at cs.nyu.edu/~taking/soi.tgz and have been submitted for inclusion into `SMT-LIB`'s `QF_LRA` family.

[7]OpenSMT gave no answer on the `latendresse` benchmarks.

Range for n	0	1	$[2, 10]$	$[11, 100]$	$[101, 1000]$	$[1000, 2238]$	total
Number of calls to SIMPLEXFORSMTCHECK with n pivots	32832424	645473	896659	174743	2362	7	34551668
$\sum$ Total number of pivots performed by these calls	0	645473	3677258	3628577	479386	10173	8440867
Number of calls to SOICHECK with n pivots	30475287	924639	1008398	130167	655	0	32539146
$\sum$ Total number of pivots performed by these calls	0	924639	3900190	2366117	126506	0	7317452

TABLE I: Number of pivots per call to check for experiment 1.

	CVC4 SOISIMPLEX		CVC4 SIMPLEXFORSMT		Z3		yices2		mathsat		opensmt	
set	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)	solved	time (s)
QFLRA (634)	627	5621.29	625	5523.15	620	5582.46	619	5300	608	8043	597	17261
miplib (48)	35	641.3	33	1760	28	1158.42	27	1616	19	3049	21	1509
latendresse (18)	18	883.53	18	205	8	17.98	10	103.38	10	94.73	-	-

TABLE II: Running time and number of problems solved for experiment 2.

VII. CONCLUSION

The authors believe these experiments demonstrate both the strength and weakness of SIMPLEXFORSMT's local optimization criteria. It is good at keeping the amount of work small in the context of a DPLL(T) style search. The local optimization criteria requires little analysis and is quite an efficient heuristic for many SMT problems; however, its global convergence is questionable on large and hard examples. SOISIMPLEX adds a global optimization criterion and appears to be more robust for large and hard examples, but this comes with the cost of some additional analysis during pivot selection. Future work will explore how to heuristically take advantage of the best characteristics of both algorithms.

ACKNOWLEDGEMENTS

We'd like to thank the other members of the NYU ACSys research group for their many contributions to CVC4. This work was funded in part by NSF Grants CCF-0644299, CNS-0917375, and NASA Cooperative Agreement NNA10DE73C.

REFERENCES

[1] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *CAV 2006*, LNCS 4144. Springer-Verlag, August 2006, pp. 81–94.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV 2011*, LNCS 6806. Springer-Verlag, 2011, pp. 171–177.

[3] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," in *TACAS 2013*, LNCS 7795. Springer-Verlag, 2013.

[4] L. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS 2008*, LNCS 4963. Springer-Verlag, 2008, pp. 337–340.

[5] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: an interpolating smt solver," in *Model Checking Software (SPIN Workshop 2012)*, LNCS 7385. Springer-Verlag, 2012, pp. 248–254.

[6] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *TACAS 2011*, LNCS 6605. Springer-Verlag, 2011, pp. 150–153.

[7] P. E. Gill, W. Murray, and M. H. Wright, *Numerical linear algebra and optimization. Vol. 1*. Redwood City, CA: Addison-Wesley Publishing Company, 1991.

[8] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[9] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, 1989.

[10] R. Niewenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)," *JACM*, vol. 53, no. 6, pp. 937–977, November 2006.

[11] B. Dutertre and L. de Moura, "Integrating Simplex with DPLL(T)," Computer Science Laboratory, SRI International, Tech. Rep. SRI-CSL-06-01, May 2006.

[12] L. de Moura, N. Bjørner, and C. Wintersteiger, "Z3 Source Code v4.3.1 `select_pivot`," http://z3.codeplex.com/SourceControl/changeset/view/89c1785b73225a1b363c0e485f854613121b70a7#src/smt/theory_arith_core.h.

[13] A. Griggio, "An Effective SMT Engine for Formal Verification," Ph.D. dissertation, DISI - University of Trento, December 2009.

[14] R. Bruttomesso, S. Fulvio Rollini, N. Sharygina, and A. Tsitovich, "OpenSMT Source Code r64," http://opensmt.googlecode.com/svn/trunk/src/tsolvers/lrasolver/LRASolver.C.

[15] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A Theorem Prover for Program Checking," *JACM*, vol. 52, no. 3, pp. 365–473, May 2005.

[16] H. Rueß and N. Shankar, "Solving linear arithmetic constraints," SRI International, Tech. Rep. SRI-CSL-04-01, 2004.

[17] G. Badros, A. Borning, and P. Stuckey, "The Cassowary linear arithmetic constraint solving algorithm," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 8, no. 4, pp. 267–306, December 2001.

[18] D. Monniaux, "On using floating-point computations to help an exact linear arithmetic decision procedure," in *CAV 2009*, LNCS 5643. Springer-Verlag, 2009, pp. 570–583.

[19] D. Caminha Barbosa de Oliveira and D. Monniaux, "Experiments on the feasibility of using a floating-point simplex in an SMT solver," in *Workshop on Practical Aspects of Automated Reasoning (PAAR)*. CEUR Workshop Proceedings, 2012.

[20] 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.

[21] U. Junker, "QuickXplain: Conflict detection for arbitrary constraint propagation algorithms," in *IJCAI-01 Workshop on Modelling and Solving Problems with Constraints*, 2001.

[22] R. B. Bruttomesso, D. Cok, and A. Griggio, "Smt-comp 2012," Jun. 2012. [Online]. Available: http://smtcomp.sourceforge.net/2012/

[23] C. Barrett, A. Stump, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2010.

[24] M. Latendresse, M. Krummenacker, M. Trupp, and P. D. Karp, "Construction and completion of flux balance models from pathway databases," *Bioinformatics*, vol. 28, p. 38896, 2012.