Efficient Term-ITE Conversion for Satisfiability Modulo Theories*

Hyondeuk Kim[1], Fabio Somenzi[1], and HoonSang Jin[2]

[1] University of Colorado at Boulder
[2] Cadence Design Systems
{Hyondeuk.Kim,Fabio}@Colorado.EDU
{hsjin}@cadence.com

Abstract. This paper describes how *term-if-then-else* (*term-ITE*) is handled in Satisfiability Modulo Theories (SMT) and to decide *Linear Arithmetic Logic* (*LA*) in particular. Term-ITEs allow one to conveniently express verification conditions; hence, they are very common in practice. However, the theory provers of SMT solvers are usually designed to work on conjunctions of literals; therefore, the input formulae are rewritten so as to eliminate term-ITEs. The challenge in rewriting is to avoid introducing too many new variables, while avoiding as often as possible the exponential explosion that is frequent when a naive approach is applied. We propose a solution that is based on cofactoring and theory propagation, which often produces orders-of-magnitude speedups in several SMT solvers for LA problems.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers find increasing applications in areas like formal verification in which one needs to reason about complex Boolean combinations of numerical constraints. The most common approach to this problem leverages the efficiency of modern propositional satisfiability solvers that work on a propositional abstraction of the given formula. At the same time, they interact with theory solvers, which check conjunctions of literals for consistency and learn consequences (new lemmas) from them. This approach has come to be known as DPLL(T) [12].

Among the logics for which theory solvers have been developed in recent times, linear arithmetic is one of the most useful and well-researched. Many current solvers adopt some variant of the simplex algorithm. In particular, the backtrackable version of [3] fits well in the DPLL(T) scheme and has shown good results in practice for both integer and real-valued variables.

The Boolean dimension of many SMT instances, however, continues to pose a challenge to solvers. In this paper we address this problem. In particular, we focus on those instances that make extensive use of the *term-if-then-else* (ITE) operator. This operator facilitates the analysis of problems in which paths through control-flow graphs must be translated into SMT formulae. It is not surprising, therefore, that many of the available benchmark instances for linear arithmetic are rich in term-ITEs. Given a code fragment

* This work was supported in part by SRC contract 1859-TJ-2008.

that contains *if* statements, a verification condition can be naturally formulated with ITEs as shown in Fig. 1.

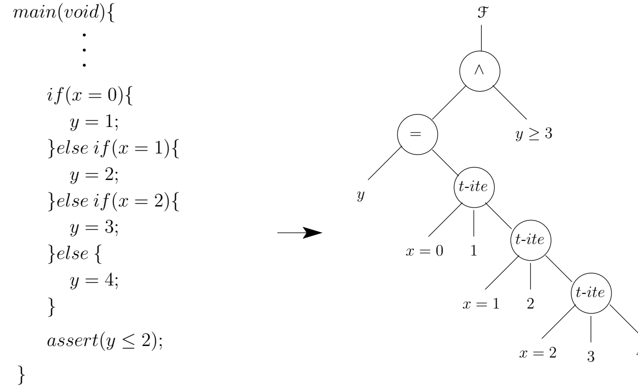

Fig. 1. Verification condition $\mathcal{F}$ with term-ITEs

Two major approaches can be envisioned to deal with term-ITEs. On the one hand, one can modify the theory solver to deal with conditional expressions. Without ITEs, every assignment to an atom of the SMT formula adds to a conjunction of literals that is analyzed by the theory solver. With ITEs, this is no longer the case. In order to analyze the atom, the conditional expressions of the ITEs need to be assigned. On the other hand, one can eliminate all the ITEs from the formula by rewriting. The problem here is that the rewritten formula may retain a lot of redundancies depending on how one rewrites it. We address this problem by a procedure based on cofactoring and theory simplification. Although our approach may cause a blow-up, it often simplifies the formula in practice. Our approach is applied to linear arithmetic logic in this paper; however, it can be easily applied to other logics like the logic of equality and uninterpreted function symbols (EUF), the logic of bit-vector, or the logic of arrays. Only the terminal cases are different in each logic. Our experiments show that our approach is promising and often speeds up a solver by orders of magnitude. The experiments also demonstrate the effectiveness of theory simplification.

The rest of this paper is organized as follows. Section 2 defines notation and summarizes the main concepts. Section 3 discusses motivation and outlines our approach to the problem. Section 4 presents the simplifications applied before invoking the term-ITE conversion. Section 5 presents an algorithm for term-ITE conversion with theory reasoning. After a survey of related work in Sect. 6, experiments are presented in Sect. 7, and conclusions are offered in Sect. 8.

2 Preliminaries

We consider the satisfiability problem for linear arithmetic logic, which is the quantifier-free fragment of first-order logic that deals with linear arithmetic constraints. Let V_B be a set of propositional variables and V_R be a set of real-valued variables. The formulae in linear arithmetic logic are inductively defined as the largest set that satisfies the following rules.

- A propositional variable $a \in V_B$ is a formula.
- A real number $c \in \mathbb{R}$ is a (constant) term.
- The product cx of a real number $c \in \mathbb{R}$ and a real-valued variable $x \in V_R$ is a term.
- If t_1 and t_2 are terms, then $t_1 + t_2$ is a term.
- If t_1 and t_2 are terms, and f is a formula, then *term-ite*(f, t_1, t_2) is a term.
- If t_1 and t_2 are terms, and $\sim$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1 \sim t_2$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae.

The semantics are defined in the usual way; in particular, *ite*(f_1, f_2, f_3) is equivalent to $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. An *atomic formula* is one of the form $t_1 \sim t_2$. A *positive literal* is an atomic formula or a propositional variable; a *negative literal* is the negation of a positive literal.

A model for a formula f is an assignment of values to the variables in the formula that is consistent with the type of each variable and that makes the formula true. A formula that has at least one model is *satisfiable*. In recent years, decision procedure for *LA*, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure. formula $\mathcal{F}$, a propositional abstraction $\mathcal{F}_b$ of $\mathcal{F}$ is built by substituting each atomic formula with a new propositional variable. As the DPLL procedure provides a model for $\mathcal{F}_b$, a *theory solver* for *LA* is invoked with the set of atomic formulae that are assigned. The theory solver checks the feasibility of the set. If the set is feasible, then the model is also a model in theory. If the set is infeasible, then the explanation of the infeasibility is returned to the DPLL procedure. The procedure continues until it finds a complete model, or decides that $\mathcal{F}$ is *unsatisfiable* in the given theory.

3 Term-ITE Conversion

An *LA* formula can often be expressed more concisely by using term-ITEs. For example, Fig. 2 shows that the formula f in (a) is equivalent to the more verbose formula f' in (b). Despite the conciseness afforded by term-ITEs, a *LA* formula with term-ITEs is often converted into a formula without them, so that the formula may be solved by an SMT solver based on the propositional abstraction.

3.1 Two Methods for Term-ITE Conversion

A common way to eliminate these term-ITEs is to introduce a fresh constant that replaces the term-ITE. In particular, an *LA* formula $f($*term-ite*$(g, t_1, t_2))$ is converted to the equisatisfiable

$$f(c) \wedge \mathit{ite}(g, t_1 = c, t_2 = c) \ , \tag{1}$$

where c is a constant that does not appear in the given formula. The advantage of this conversion is that it does not blow up; however, it often retains redundancies in the converted formula. For example, the formula *term-ite*$(g, 1, 2) =$ *term-ite*$(h, 3, 4)$ can be reduced to $\perp$, whereas the conversion generates *ite*$(g, c = 1, c = 2) \wedge$ *ite*$(h, c = 3, c = 4)$ that contains a redundancy. To remove the redundancy, additional theory reasoning is required. A naive approach to the term-ITE conversion will be to combine every term in the left-hand side of the relational operator with the terms in the right-hand side depending on the conditional terms of term-ITEs. In particular, an *LA* formula $f($*term-ite*$(g, t_1, t_2))$ is converted according to following conversion rule [7].

$$f(\textit{term-ite}(g, t_1, t_2)) \iff \textit{ite}(g, f(t_1), f(t_2)) \ . \tag{2}$$

This approach removes the redundancy in the above example on the fly; however, as Fig. 2 illustrates, the converted formula may grow exponentially large in the worst case.

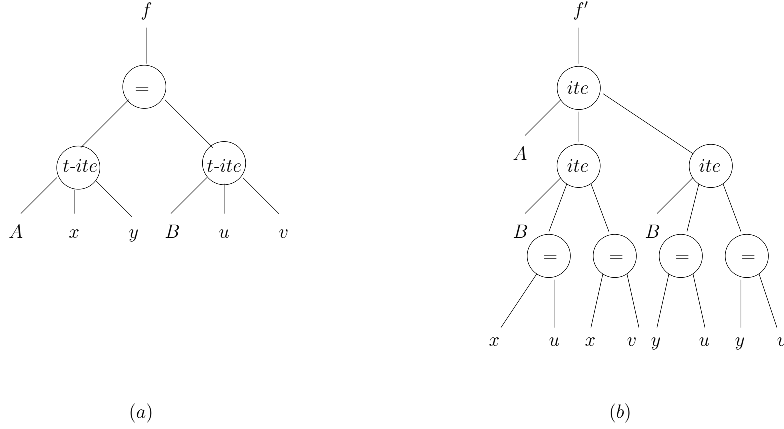

Fig. 2. Term-ITE conversion

3.2 Term-ITE Conversion with Cofactors

As an alternative to the approaches described in Sect. 3.1, term-ITE conversion can be done by computing cofactors.

Definition 1. *Let $f(x_1, ..., x_n)$ be an LA formula, where each x_i is a positive literal. Then,*

$$f_{x_i} = f(x_1, ..., x_{i-1}, \top, x_{i+1}, ..., x_n)$$
$$f_{\neg x_i} = f(x_1, ..., x_{i-1}, \perp, x_{i+1}, ..., x_n)$$

are the positive and negative cofactors of f with respect to x_i.

Theorem 1 (Boole). *Let $f(x_1, ..., x_n)$ be an LA formula. Then $f(x_1, ..., x_n) = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i}) = \mathit{ite}(x_i, f_{x_i}, f_{\neg x_i})$.*

According to Theorem 1, the following rule can be used to rewrite an *LA* formula:

$$f(\textit{term-ite}(g, t_1, t_2)) \iff \mathit{ite}(x, f_x(\textit{term-ite}(g, t_1, t_2)), f_{\neg x}(\textit{term-ite}(g, t_1, t_2))) \ . \tag{3}$$

By computing the cofactors of f, the conversion may greatly simplify the converted formula. In Fig. 3, f is simplified to $\bot$ using (3). In particular, the cofactors $f_A \iff (\textit{term-ite}(B, 3, 5) = 4)$ and $f_{\neg A} \iff (5 = 4) \iff \bot$ are first computed. Then f is simplified to $(A \wedge f_A)$, and finally reduced to $\bot$ by cofactoring f_A with respect to B.

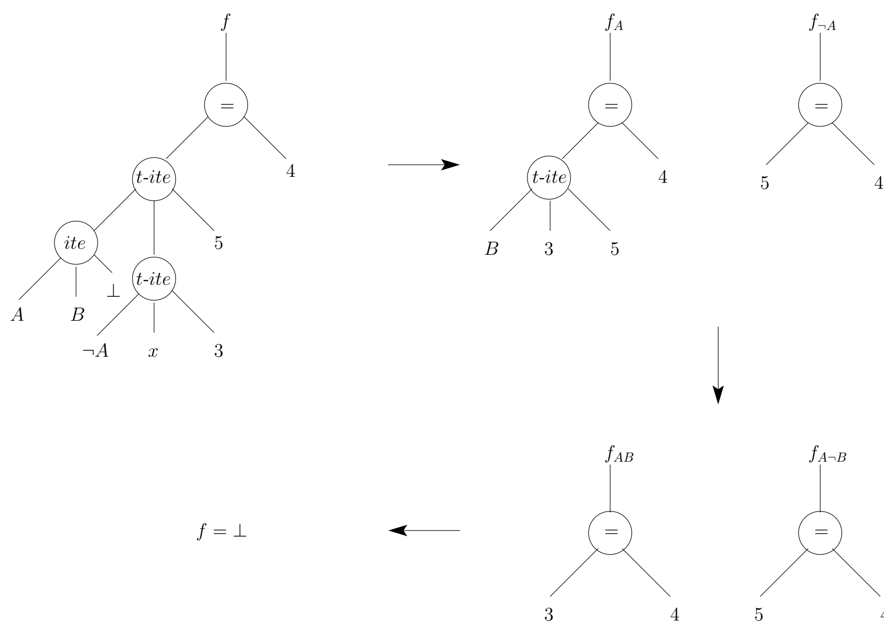

Fig. 3. Term-ITE conversion with cofactor

This kind of simplification can often be applied to the *LA* problems in SMT-LIB [14]. As the previous example shows, the simplification for equality is easily done by comparing two constants. On the other hand, if fresh constants are introduced, redundancy may remain in the converted formula: a fresh constant c replaces the term $\textit{term-ite}(\mathit{ite}(A, B, \bot), \textit{term-ite}(\neg A, x, 3), 5)$ in f. Then f is rewritten in two steps: first as

$$(c = 4) \wedge \mathit{ite}(\mathit{ite}(A, B, \bot), c = \textit{term-ite}(\neg A, x, 3), c = 5) \ ,$$

and then as

$$(c = 4) \wedge (c' = c) \wedge \mathit{ite}(\mathit{ite}(A, B, \bot), \mathit{ite}(\neg A, c' = x, c' = 3), c = 5) \ ,$$

where c' is another fresh constant. Removing the redundancy from the converted formula requires theory reasoning. While such reasoning is uncomplicated in this example, in general the new constants may make it cumbersome. Although the cofactoring method may give a huge reduction, it may blow up if there is little simplification. Compared to the approach that introduces a fresh constant, it is more aggressive.

Definition 2. *Let x be a literal and h be a formula. We write $x \models_T h$ if h is a consequence of x in theory T, and we call h a* theory consequence *of x.*

The cofactoring method can be further extended with theory reasoning. Using the theory propagation method [12], an assignment to an atomic predicate may entail assignments to other atomic predicates. For example, in *LA*, if we make an assignment to $(x < 0) = \top$, then $(x < 3) = \top$ and $(x > 1) = \bot$. The following rules show how theory propagation may help in the simplification of the converted formula:

$$\frac{x \models_T h}{f_x(\mathit{term\text{-}ite}(h, t_1, t_2)) \iff f_x(t_1)} \quad (4)$$

$$\frac{x \models_T \neg h}{f_x(\mathit{term\text{-}ite}(h, t_1, t_2)) \iff f_x(t_2)} \ . \quad (5)$$

As we compute the cofactors in the term-ITE conversion, we make an assignment to the cofactoring literal. If the cofactoring literal is an atomic formula and the computed cofactor is also an atomic formula, then theory reasoning can be invoked to check the relation between these two atoms. The following consequence of Theorem 1 gives an idea of how this simplification can be done; it will be used in Sect. 5.

Theorem 2. *Given a formula f of theory T and a literal x_i, if $x_i \models_T f_{x_i}$, then $f \iff x_i \vee f_{\neg x_i}$. If $x_i \models_T \neg f_{x_i}$, then $f \iff \neg x_i \wedge f_{\neg x_i}$.*

4 Simple Preprocessing

Before we execute term-ITE conversion for an *LA* formula f, terminal cases for term-ITE are detected and basic simplification is carried out. Let $a \in V_B$; let t_1, t_2, and t_3 be terms and let c_1, c_2, and c_3 be constants. In the *LA* formula, we detect special cases like $\mathit{term\text{-}ite}(\top, t_1, t_2) \iff t_1$, $\mathit{term\text{-}ite}(\bot, t_1, t_2) \iff t_2$, $\mathit{term\text{-}ite}(a, t_1, t_1) \iff t_1$. We also simplify nested term-ITEs such as $\mathit{term\text{-}ite}(a, \mathit{term\text{-}ite}(a, t_1, t_3), t_2) \iff \mathit{term\text{-}ite}(a, t_1, t_2)$, $\mathit{term\text{-}ite}(a, \mathit{term\text{-}ite}(\neg a, t_3, t_2), t_1) \iff \mathit{term\text{-}ite}(a, t_2, t_1)$. For arithmetic terms, $(0 + t_1) \iff t_1, (0 \cdot t_1) \iff 0, (1 \cdot t_1) \iff t_1, (-(-t_1)) \iff t_1, (c_1 + c_2) \iff c_3$, where c_3 is the sum of c_1 and c_2.

Furthermore, if a formula f has a root node that is a relational operator applied to term-ITEs and has leaves that are all constants, then it can be simplified. For simplicity, we only check the case where either of the children of the root node is a constant. Example 1 shows such a case.

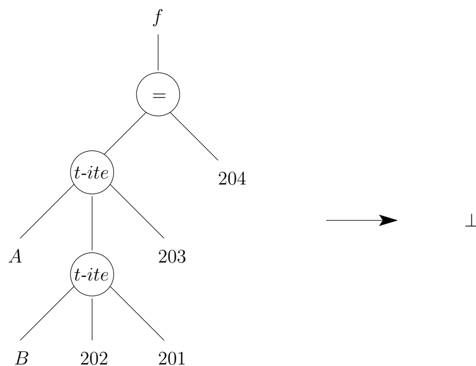

Fig. 4. Term-ITE conversion with simple check

Example 1. Let f be a formula shown in Fig. 4. The formula f is an equality with term-ITEs. As Fig. 4 shows, the terms on the left-hand side of the root node are all constants and the one on the right-hand side is also a constant. In such a case, we compare all the constants in the left hand side for equality with the constant on the right, 204. Clearly, $(202 = 204) \iff \bot$, $(201 = 204) \iff \bot$ and $(201 = 203) \iff \bot$; hence $f = \bot$.

5 Algorithm

We assume that an SMT solver adopts the rewriting procedure. Given an *LA* formula $\mathcal{F}$ with term-ITEs, an SMT solver converts $\mathcal{F}$ into $\mathcal{F}'$ by removing all term-ITEs in $\mathcal{F}$. The SMT solver then decides the satisfiability of $\mathcal{F}'$. In this section, we describe how $\mathcal{F}$ is converted into $\mathcal{F}'$.

As the pseudocode of Fig. 5 shows, the main function of term-ITE conversion is called with an *LA* formula $\mathcal{F}$. The formula $\mathcal{F}$ is represented as a directed acyclic graph (DAG), where each node is a Boolean operator, a relational operator, an arithmetic operator, a term-ITE, or an atom. The conversion is applied to each relational operator in the DAG, and the procedure ends when $\mathcal{F}'$ no longer contains term-ITEs. The main function starts by selecting the candidates for the conversion in the DAG. Each candidate is a relational operator that has a term-ITE as a descendant, and the candidates are gathered in F. As Line 4 in Fig. 5 shows, the term-ITE conversion is invoked with $f \in F$, and all the term-ITEs are removed from f. After the conversion of f, the converted formula f' is either a Boolean ITE or an atom. The procedure ends when all $f \in F$ have been considered. At that point, $\mathcal{F}$ has been converted into $\mathcal{F}'$, which does not contain any term-ITEs.

As TermIteConversion is invoked with $f \in F$, a cofactoring variable v is searched for in f at Line 10. We select an atom as a cofactoring variable that resides in the conditional term of the term-ITE. With v, we recursively compute the cofactor of f. In general, the cofactors are computed for the children of f with respect to v, and a

```
1   TermIteConversionMain (𝓕) {
2       F := GatherCandidateForTermIteConversion (𝓕);
3       for (each f ∈ F in topological order) {
4           f′ := TermIteConversion (f);
5           𝓕′ := UpdateFormula (𝓕, f′);
6       }
7       return 𝓕′;
8   }

9   TermIteConversion (f) {
10      while ( v := GetCofactorVariable (f) ) {
11          f_v := CofactorRecur (f, v);
12          f_¬v := CofactorRecur (f, ¬v);
13          f := Ite (v, f_v, f_¬v);
14      }
15      return f;
16  }

17  CofactorRecur (f, v) {
18      if ( f = v ) {
19          f_v := ⊤;
20      } else if ( f = ¬v ) {
21          f_v := ⊥;
22      } else if ( is_relation(f) ) {
23          f_v := CofactorRelRecur (f, v);
24      } else if ( is_term_ite(f) ) {
25          f_v := CofactorTiteRecur (f, v);
26      } else { /* +, −, × */
27          C := children(f);
28          For each c ∈ C {
29              d := CofactorRecur (c, v);
30              Add(D, d);
31          }
32          f_v := NewFormula (type(f), D); /* type(f) is either +, −, ×. */
33          SimplifyArithFormula(f_v);
34      }
35      return f_v;
36  }
```

Fig. 5. Term-ITE conversion algorithm

new formula f_v is created with new children. As shown in Line 38 of Fig. 6, if f is a relational operator, we compute the cofactors l_v and r_v for the children of f. After computing the cofactors, we check for simple cases with l_v and r_v. The simple check detects terminal cases for the terms l_v and r_v with respect to the type $(=, <, \leq, >, \geq)$

```
37  CofactorRelRecur (f, v) {
38      l_v := CofactorRelRecur (left(f), v);
39      r_v := CofactorRelRecur (right(f), v);
40      f_v := SimpleCheckWithTerms (type(f), l_v, r_v);
41      if ( f_v = NoSimplification ) {
42          f_v := NewFormula (type(f), l_v, r_v);
43          if ( is_term_ite(l_v) or is_term_ite(r_v) ) {
44              f_v = TermIteConversion (f_v);
45          }
46      }
47      if ( is_atom(f_v) ) {
48          if ( v ⊨_T f_v ) { /* theory reasoning */
49              f_v := ⊤
50          } else if ( v ⊨_T ¬f_v ) { /* theory reasoning */
51              f_v := ⊥
52          }
53      }
54      return f_v;
55  }

56  CofactorTiteRecur (f, v) {
57      f_c := CondTerm(f);  f_t := ThenTerm(f);  f_e := ElseTerm(f);
58      if ( f_c = ⊤ ) {
59          return CofactorRecur (f_t, v);
60      } else if ( f_c = ⊥ ) {
61          return CofactorRecur (f_e, v);
62      } else if ( is_pred(f_c) ) {
63          if ( v ⊨_T f_c ) { /* theory reasoning */
64              return CofactorRecur (f_t, v);
65          } else if ( v ⊨_T ¬f_c) ) { /* theory reasoning */
66              return CofactorRecur (f_e, v);
67          }
68      }
69      c_v := CofactorRecur (f_c, v);
70      t_v := CofactorRecur (f_t, v);
71      e_v := CofactorRecur (f_e, v);
72      f_v := Ite (c_v, t_v, e_v);
73      return f_v;
74  }
```

Fig. 6. Term-ITE conversion algorithm

of f. Figure 4 shows an example of simplification. If a terminal case is not found, a new formula f_v is generated with $\mathrm{type}(f)$, l_v and r_v. The newly generated formula, f_v is either an atom or a relation operator with term-ITEs. In the latter case, term-ITE conversion is called with f_v, again. In Line 47 of Fig. 6, if f_v is an atom, theory

reasoning is done with v. As Theorem 2 shows, if $v \models_T f_v$, then f in Line 13 of Fig. 5 is simplified to $v \vee f_{\neg v}$. Likewise, if $v \models_T \neg f_v$, then f is simplified to $\neg v \wedge f_{\neg v}$. When f is either a term-ITE or a Boolean ITE, the cofactor for each term of f is computed as shown in Line 58 of Fig. 6. As in the cofactoring on the relational operator, a terminal case is checked for the conditional term f_c. If f_c is an atomic predicate, theory reasoning is done with v and f_c using Rules 4–5 of Sect. 3.2. If a terminal case is not found, then the cofactors for the terms of f are computed to obtain f_v.

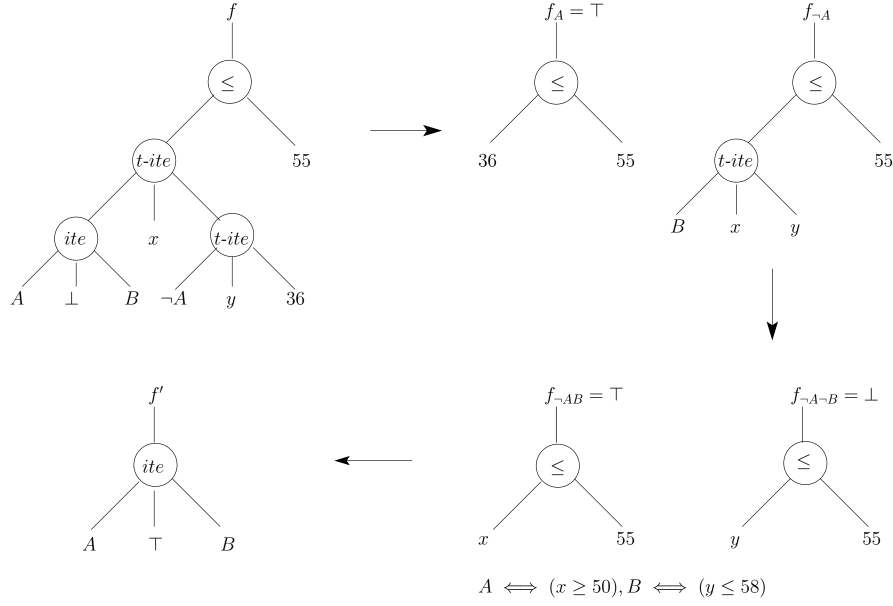

Fig. 7. Term-ITE conversion

Example 2. If f is a relational operator such that $D(f)$ contains term-ITEs, we convert f into f' such that there is no term-ITE in $D(f')$. In Fig. 7, let $A \leftrightarrow (x \geq 50)$ and $B \leftrightarrow (y \leq 58)$. We first traverse $D(f)$ to find a cofactoring variable. We pick an atomic formula A as cofactoring variable and compute the cofactors of f with respect to A. As we proceed, $f_A = (36 \leq 55) = \top$ and $f_{\neg A}$ is constructed with a new term-ITE. Since there still exists a term-ITE in $D(f_{\neg A})$, we look for another cofactoring variable in $f_{\neg A}$. We select B and compute the cofactors for $f_{\neg A}$. As a result, we get $f_{\neg AB} = (x \leq 55)$ and $f_{\neg A \neg B} = (y \leq 55)$. Since $A \models_T f_{\neg AB}$ and $\neg B \models_T \neg f_{\neg A \neg B}$, $f_{\neg AB} = \top$ and $f_{\neg A \neg B} = \bot$. Finally, the converted formula f' gets reduced to $ite(A, \top, B)$ as shown in Fig. 7.

6 Related Work

Early references on the treatment of ITEs are [8], [2] and [7]. For SMT preprocessing, HTP [13] introduces several preprocessing techniques such as unate predicate detection, variable substitution and symmetry breaking. Yices [3] uses a Gaussian elimination to reduce the size of initial tableau of equality constraints. In [17], Yu *et al.* describes a static learning technique that analyzes the relationship of the linear constraints. In Karplus's technical report [8], a new canonical form for *ITE* DAGs is introduced using two-cuts, and *ITE* normalization using recursive transformation is shown in [11].

7 Experimental Results

We have implemented the algorithm presented in Sect. 5 in Sateen [10, 9, 15], a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [5, 6] with theory-specific procedures. Experiments are done with the full set of QF_LIA (Quantifier free linear integer arithmetic logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition) [14]. The experiments were performed on an Intel 2.4 GHz Quad Core with 4 GB of RAM running Linux. Time out was set at 1000 seconds. Sateen was compared with Z3.2 [14], MathSAT-4.2[1, 14] and Yices-1.0.16 [16]. Z3.2 and MathSAT-4.2 are the ones that were submitted to SMT-COMP in 2008. We used most recent version of Yices that is available.

In QF_LIA benchmarks, there are two benchmark sets, *nec-smt* and *rings*, that are rich in term-ITE operators. More than 90 percent of the QF_LIA benchmarks belong to those two sets. The instances in the *nec-smt* set are generated by the SMT-based BMC engine of F-Soft [4]; the instances in *rings* encode associativity properties on modular arithmetic.

Figures 8–10 show scatterplots comparing Z3, MathSAT and Yices to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 8 shows that Sateen is often an order of magnitude faster than Z3. In Fig. 9 and 10, Sateen is often a few orders of magnitude faster than MathSAT and Yices.

We further evaluated our preprocessor by generating simplified formulae from the *nec-smt* benchmarks and running Z3, MathSAT, and Yices on them. All solvers took less than a second on each simplified problem. Figures 11–13 show scatterplots comparing Z3, MathSAT and Yices with preprocessor and without preprocessor. The times for the solvers with preprocessor include preprocessing time. As Figures 11–13 show, our preprocessor is also effective for other solvers.

Table 1 shows the number of term-ITE reductions with the simple preprocessing on randomly selected benchmarks. The first column gives the name of the benchmarks, the second one is the initial number of term-ITEs, and the third one is the number of term-ITEs after the simple preprocessing. The last column gives the rate of the reduction. On average, we achieved 15% term-ITE reduction with the simple preprocessing of Section 4.

Finally, we compared our approach to the naive approach of Eq. 2. As Fig. 15 shows, our approach is significantly better. In addition, we disabled theory simplification in the

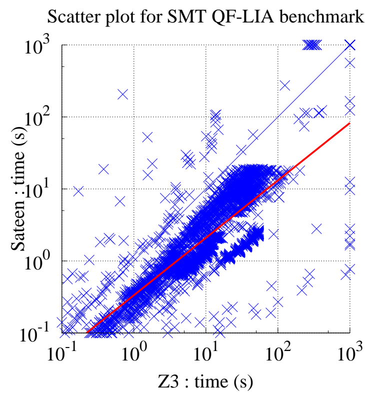

Fig. 8. Z3 vs. Sateen on QF_LIA

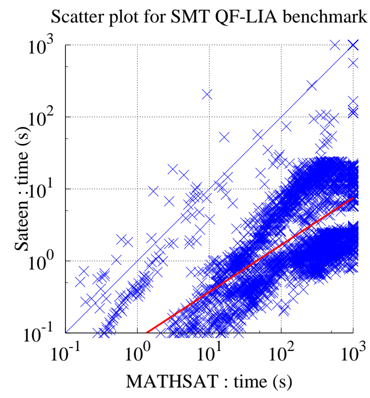

Fig. 9. MATHSAT vs. Sateen on QF_LIA

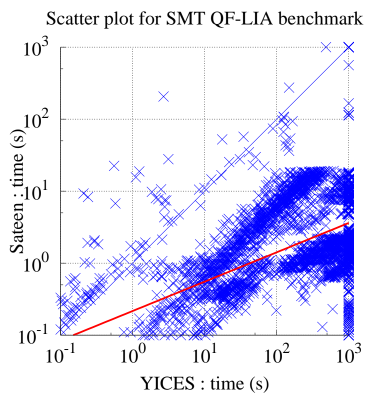

Fig. 10. YICES vs. Sateen on QF_LIA

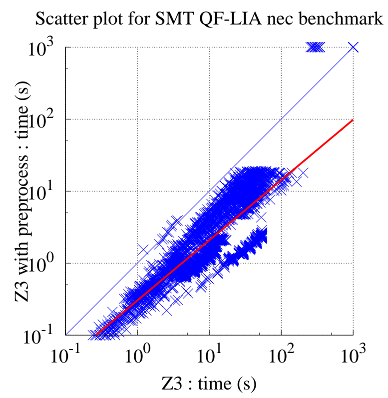

Fig. 11. Z3 WITH PREPROCESS vs. Z3 on QF_LIA

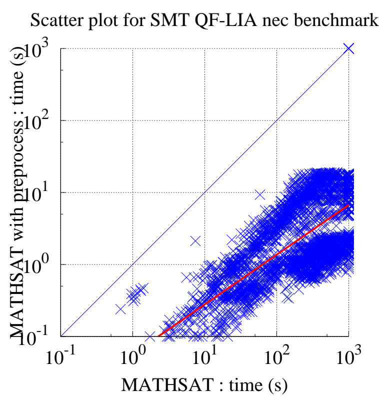

Fig. 12. MATHSAT WITH PREPROCESS vs. MATHSAT on QF_LIA

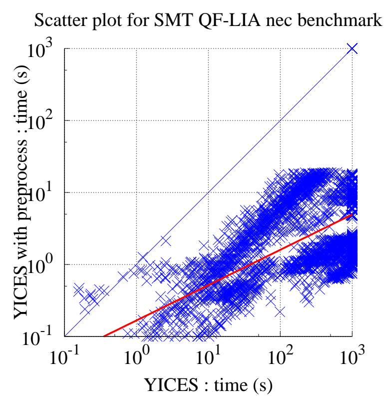

Fig. 13. YICES WITH PREPROCESS vs. YICES on QF_LIA

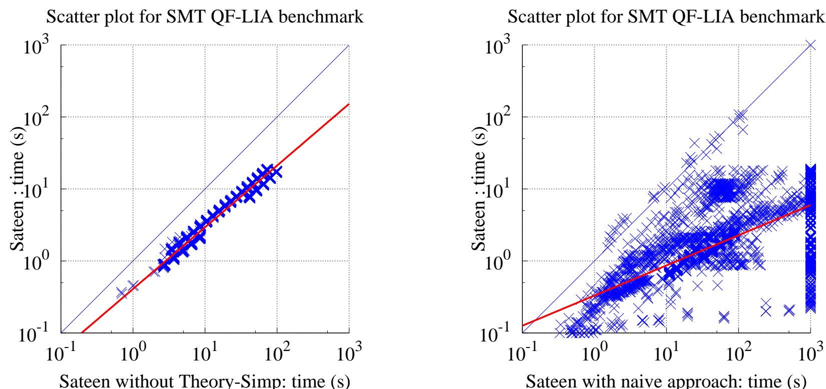

Fig. 14. SATEEN vs. Sateen without Theory-Simp on QF_LIA

Fig. 15. SATEEN vs. Sateen with naive approach on QF_LIA

Table 1. Number of term-ITE reduction with simple preprocessing

Benchmark	**Before S.P.**	**After S.P.**	**rate(%)**
bftpd_login/prp-74-50.smt	38773	34085	12
checkpass/prp-10-46.smt	17240	14949	13
checkpass/prp-63-50.smt	25376	21893	14
checkpass_pwd/prp-38-42.smt	12196	10354	15
getoption/prp-2-200.smt	11269	9791	13
getoption_directories/prp-0-110.smt	72892	62457	14
getoption_group/prp-72-49.smt	15021	12094	20
handler_sigchld/prp-20-46.smt	7800	6824	13
int_from_list/prp-34-41.smt	7184	5888	18
user_is_in_group/prp-23-48.smt	22549	17939	20

algorithm and ran the experiment on the problems where the simplifications play a significant role. Figure 14 shows that Sateen with theory simplification is consistently better than the one without simplification.

8 Conclusions

We have presented an algorithm for the term-ITE conversion in first-order theories like the theory of linear arithmetic. The approach is based on the computation of cofactors and theory simplification. The simplification is done by detecting special cases in the formula or using theory propagation on the atomic predicates. Experiments show that this approach is very effective in most QF_LIA benchmarks and often speeds up SMT solvers. On the other hand, since our approach may still blow up in general, we are working on combining it with a less aggressive approach, based on (1), that does not blow up.

Acknowledgment. The authors thank the reviewers for their detailed suggestions.

References

[1] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, pages 317–333, Edinburgh, UK, Apr. 2005. LNCS 3440.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, Orlando, FL, June 1990.

[3] B. Duterte and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Eighteenth Conference on Computer Aided Verification (CAV'06)*, pages 81–94, Seattle, WA, Aug. 2006. LNCS 4144.

[4] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *17th International Conference on Computer-Aided Verification(CAV)*, pages 301–306, 2005.

[5] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, pages 287–300, Apr. 2005. LNCS 3440.

[6] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In *Proceedings of the Design Automation Conference*, pages 750–753, Anaheim, CA, June 2005.

[7] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *Proceedings of the International Conference on Computer-Aided Design*, pages 2–6, San Jose, CA, Nov. 1995.

[8] K. Karplus. Representing Boolean functions with if-then-else DAGs. In *Technical Report UCSC-CRL-88-28, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064*, Dec. 1988.

[9] H. Kim, H. Jin, K. Ravi, P. Spacek, J. Pierce, B. Kurshan, and F. Somenzi. Application of formal word-level analysis to constrained random simulation. In *20th International Conference on Computer Aided Verification (CAV'08)*, July 2008.

[10] H. Kim, H. Jin, and F. Somenzi. Disequality management in integer difference logic via finite instantiations. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:47–66, 2007.

[11] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. In *ACM Transactions on Programming Languages and Systems, 1(2):245-257*, Oct. 2008.

[12] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 321–334. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[13] K. Roe. The heuristic theorem prover: Yet another SMT modulo theorem prover. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 467–470, 2006.

[14] URL: http://smtcomp.org/.

[15] URL: http://vlsi.colorado.edu/∼vis.

[16] URL: http://yices.csl.sri.com.

[17] Y. Yu and S. Malik. Lemma learning in SMT on linear constraints. In A. Biere and C. P. Gomes, editors, *Proceedings of Theory and Applications of Satisfiability Testing – SAT 2006*, pages 142–155, Aug. 2006.