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 *Linear Arithmetic Logic* (*LA*) problem. In *LA* problem, *Term-ITE* is used to express a set of atomic formulae with the same relational operator type ($<, \leq, >, \geq$) and a different set of terms. To handle *LA* problem with (*Term-ITE*) in *SMT* solver, *term-if-then-else* (*Term-ITE*) to Boolean *if-then-else* (*ITE*) conversion is required. The conversion induces an exponential blow-up in worst case. We show how effectively *Term-ITE* is handled in *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) [11].

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

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,

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

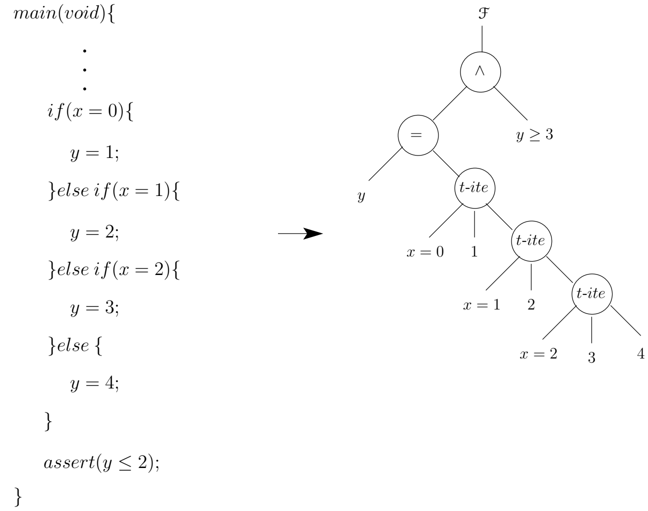

Fig. 1. Verification condition $\mathcal{F}$ with Term-Ites

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, the logic of array, etc. Only the terminal cases are different in each logic. Our experiments show that our approach is promising and often speeds up a solver by a few 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 of *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 is a term, $r \in \mathbb{R}$ is a real number, and $\sim \in \{=, \neq, <, \leq\}$ is a relational operator, then $t \sim r$ is a formula.
- If f_1, f_2 and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$ and *ite*(f_1, f_2, f_3) are formulae.

Further types of formulae can be defined as abbreviations. For instance, $t \neq c$ is defined as $\neg(t = c)$ and $a \vee b$ as $\neg(\neg a \wedge \neg b)$. An *atomic formula* is one of the form $t \sim c$, where t is a term and c is a constant. A *positive literal* is either a propositional variable or an atomic formula; a *negative literal* is the negation of a positive literal. A *clause* is the disjunction of a set of literals such that no two literals in the set are identical or complementary. A formula is in *conjunctive normal form* (CNF) if it is the conjunction of a set of clauses.

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

3.1 Term-ITE

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 formula f' in (b), but is more concise. Despite the conciseness of *term-ite* representation, *LA* formula with *term-ites* are often converted into a formula without these *term-ites*, so that the formula may be solved by an SMT solver based on the propositional abstraction. 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 $f(c) \wedge$ *if-then-else*$(g, t_1 = c, t_2 = c)$ 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 $\bot$, whereas the conversion generates a rather complex formula *if-then-else*$(g, c = 1, c = 2) \wedge$ *if-then-else*$(h, c = 3, c = 4)$ 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.

$$f(\textit{term-ite}(g, t_1, t_2)) \iff (g \wedge f(t_1)) \vee (\neg g \wedge f(t_2))$$

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

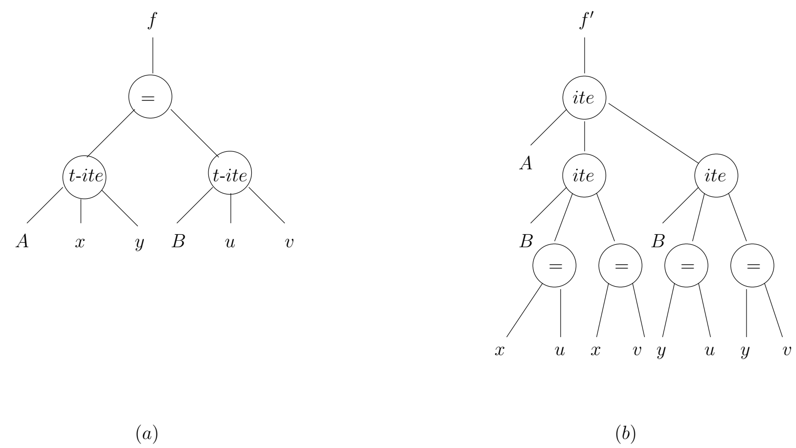

Fig. 2. Term-Ite conversion

3.2 Term-ITE Conversion with Cofactors

As an alternative to the naive 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 an atomic predicate. Then,*

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

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

Theorem 1 (Boole). *Let $f(x_1, ..., x_n)$ be a Boolean function. Then $f(x_1, ..., x_n) = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i})$.*

According to Theorem 1, an *LA* formula $f(\textit{term-ite}(g, t_1, t_2))$ can be rewritten as

$$(g \wedge f_g(t_1)) \vee (\neg g \wedge f_{\neg g}(t_2)). \tag{1}$$

By computing the cofactors for f, the conversion gets a great benefit of simplifying the converted formula. In Fig. 3, f is simplified to $\bot$ using the conversion rule. 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 for the conversion $f \iff (A \wedge f_A) \vee (\neg A \wedge f_{\neg A})$. Then f is simplified to $(A \wedge f_A)$, and finally reduced to $\bot$ by cofactoring f_A with respect to B. In practice, this kind of simplification can be often done in *LA* problems of SMT-LIB [13]. As the conversion shows, the simplification for equality is easily done by comparing two constants. On the other hand, if we use the conversion that introduces a fresh constant, the redundancy still resides in the converted formula. Following the conventional conversion rule, $\textit{term-ite}(\textit{ite}(A, B, \bot), \textit{term-ite}(\neg A, x, 3), 5)$ in f is replace with a fresh constant c. Then f is converted to

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

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

To remove the redundancy in the converted formula, a rather complicated theory reasoning is required. Although the cofactoring method gives a huge reduction, it may still blow up if there is no simplification. Compared to the approach that introduces a fresh constant, our approach is more aggressive.

Definition 2. *Let x_1 and x_2 be atomic formulae. We write $x_1 \models_T x_2$ if x_2 is a consequence of x_1 in theory T, and we call x_2 a* theory consequence *of x_1.*

In *LA*, the cofactoring method can be further extended with theory reasoning. Using the theory propagation method [11], an assignment to an atomic predicate may entail the assignments to other atomic predicates. For example, if we make an assignment to $(x < 0) = \top$, then $(x < 3) = \top$ and $(x > 1) = \bot$. Following rules give how theory propagation helps to simplify the converted formula.

$$\frac{g \models_T h}{f_g(\textit{term-ite}(h, t_1, t_2)) \iff f_g(t_1)} \tag{2}$$

$$\frac{g \models_T \neg h}{f_g(\textit{term-ite}(h, t_1, t_2)) \iff f_g(t_2)} \tag{3}$$

As we compute the cofactors in *Term-ite* conversion, we make an assignment to a cofactoring variable. If the cofactoring variable is an atomic predicate and the computed cofactor is also an atomic predicate, then the theory reasoning can be invoked to check the relation between these two atoms. The following theorem gives an idea of how this simplification can be done, and it will be used in Sect. 5.

Theorem 2. *Given an LA formula f and an atomic predicate x_i, if $x_i \models_T f_{x_i}$, then $f = x_i \vee f_{\neg x_i}$. If $x_i \models_T \neg f_{x_i}$, then $f = \neg x_i \wedge f_{\neg x_i}$.*

Proof. By Theorem 1. □

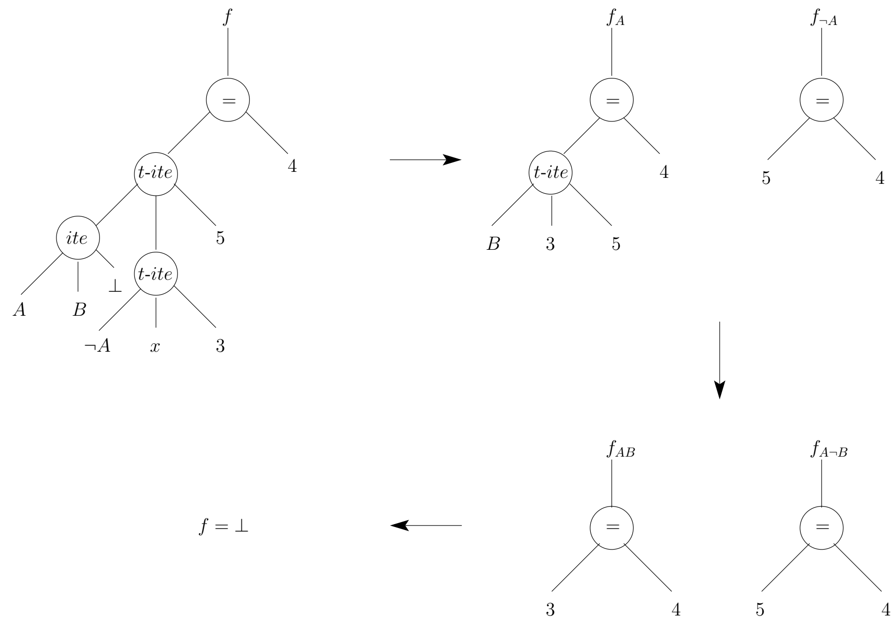

Fig. 3. Term-ITE conversion with cofactor

4 Simple Preprocessing

Before we execute *Term-Ite* conversion for an *LA* formula $\mathcal{F}$, terminal cases for *term-ite* are detected and basic simplification is done for the formula. Let $a \in V_B$; let t_1, t_2, and t_3 be terms and each $c_1, c_2, c_3 \in \mathbb{R}$. In the *LA* formula, we detect terminal cases like *term-ite*$(\top, t_1, t_2) = t_1$, *term-ite*$(\bot, t_1, t_2) = t_2$, *term-ite*$(a, t_1, t_1) = t_1$. We also simplify nested *term-ites* such as *term-ite*$(a,$ *term-ite*$(a, t_1, t_3), t_2)$ = *term-ite*(a, t_1, t_2), *term-ite*$(a,$ *term-ite*$(\neg a, t_3, t_2), t_1)$ = *term-ite*(a, t_2, t_1). For arithmetic terms, $(0+t_1) = t_1, (0 * t_1) = 0, (1 * t_1) = t_1, (-(-t_1)) = t_1, (c_1 + c_2) = c_3$.

Furthermore, if a formula f has a root node that is a relational operator with *term-ites* and has leaves that are all constants, then it can be simplified. Example 1 shows such a case.

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 $f = \bot$.

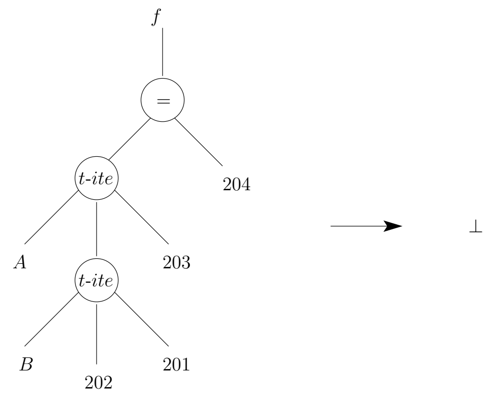

Fig. 4. Term-Ite conversion with simple check

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 $\mathcal{F}$. After the conversion, the SMT solver decides the satisfiability of $\mathcal{F}'$. In this section, we describe how $\mathcal{F}$ is converted into $\mathcal{F}'$.

As the pseudocode shows in Fig. 5, the main function of *Term-Ite* conversion is called with an *LA* formula $\mathcal{F}$. The formula $\mathcal{F}$ is represented as a *DAG (directed acyclic graph)*, 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 if $\mathcal{F}$ no longer has *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 is continued until all $f \in F$ are considered. When the *Term-Ite* conversion finishes, $\mathcal{F}$ has been converted into $\mathcal{F}'$, and $\mathcal{F}'$ does not contain any *term-ites*.

As *Term-Ite* conversion is invoked with $f \in F$, a cofactoring variable v is searched for in f in 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 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 case with l_v and r_v. The simple check detects a terminal case for the terms l_v and r_v with respect to the type $(=, <, \leq, >, \geq)$ of f. Figure 4 shows an example of simplification. If the terminal case is not found, a new formula f_v is generated with $type(f)$, 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.

```
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

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

```
CofactorRelRecur ($f$, $v$) {
    $l_v$ := CofactorRelRecur ($f \rightarrow left$, $v$);
    $r_v$ := CofactorRelRecur ($f \rightarrow right$, $v$);
    $f_v$ := SimpleCheckWithTerms (type($f$), $l_v$, $r_v$);
    if ( $f_v$ = 0 ) { /* $f_v$ is either an atom or 0 */
        $f_v$ := NewFormula (type($f$), $l_v$, $r_v$);
        if ( is_term_ite($l_v$) $or$ is_term_ite($r_v$) ) {
            $f_v$ = TermIteConversion ($f_v$);
        }
    }
    if ( is_pred($f_v$) ) {
        if ( $v \models_T f_v$ ) { /* theory reasoning */
            $f_v$ := $\top$
        } else if ( $v \models_T \neg f_v$ ) { /* theory reasoning */
            $f_v$ := $\bot$
        }
    }
    return $f_v$;
}

CofactorTiteRecur ($f$, $v$) {
    $f_c$ := CondTerm($f$); $f_t$ := ThenTerm($f$); $f_e$ := ElseTerm($f$);
    if ( $f_c$ = $\top$ ) {
        return CofactorRecur ($f_t$, $v$);
    } else if ( $f_c$ = $\bot$ ) {
        return CofactorRecur ($f_e$, $v$);
    } else if ( is_pred($f_c$) ) {
        if ( $v \models_T f_c$ ) { /* theory reasoning */
            return CofactorRecur ($f_t$, $v$);
        } else if ( $v \models_T \neg f_c$) ) { /* theory reasoning */
            return CofactorRecur ($f_e$, $v$);
        }
    }
    $c_v$ := CofactorRecur ($f_c$, $v$);
    $t_v$ := CofactorRecur ($f_t$, $v$);
    $e_v$ := CofactorRecur ($f_e$, $v$);
    $f_v$ := Ite ($c_v$, $t_v$, $e_v$);
    return $f_v$;
}
```

Fig. 6. Term-Ite conversion algorithm

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 the Rules 2–3 in Sect. 3.2. If the terminal case is not found, then the cofactors for the terms of f are computed to obtain f_v.

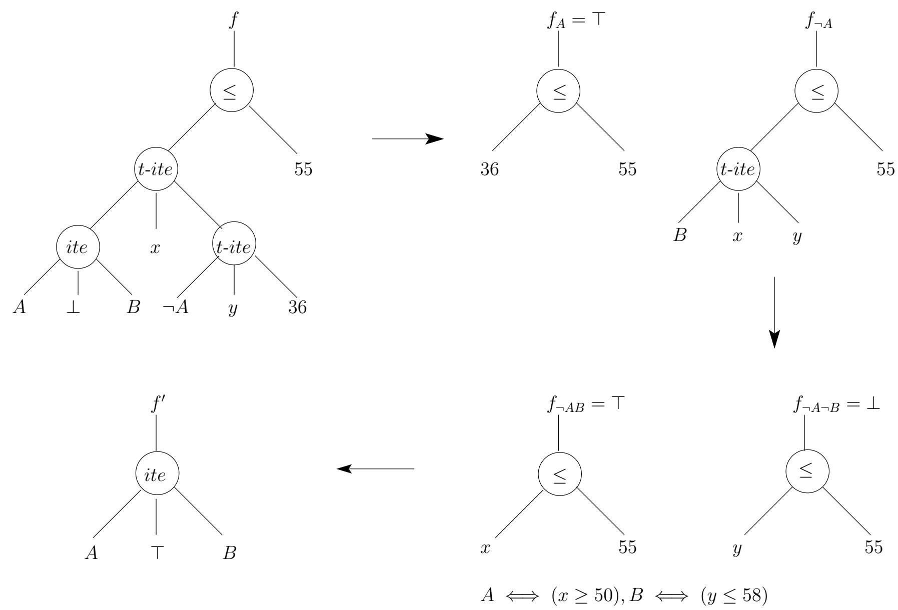

Fig. 7. Term-Ite conversion

Example 2. Let 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 a cofactoring variable and compute cofactors for 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 the Fig. 7 shows.

6 Related Work

In recent years, a number of decision procedures for *LA* have been proposed. Yices [2] presented a new Simplex-based linear arithmetic solver that enables fast backtracking and efficient integration with DPLL(T) framework. MathSAT [1] introduced a lazy and layered approach, and BarcelogicTools [11] presented DPLL(T) with exhaustive theory propagation.

For SMT preprocessing, HTP [12] introduces several preprocessing techniques such as unate predicate detection, variable substitution and symmetry breaking. Yices [2] uses a Gaussian elimination to reduce the size of initial tableau of equality constraints. In [15], Yu *et al.* describes a static learning technique that analyzes the relationship of the linear constraints. In Karplus's technical report [6], a new canonical form for *ITE* DAGs is introduced using two-cuts, and *ITE* normalization using recursive transformation is shown in [10].

7 Experimental Results

We have implemented the algorithm presented in Sect. 5 in Sateen [9, 8, 7], a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [4, 5] 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) [13]. The experiments were performed on a 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 [13], MathSAT-4.2[13] and Yices-1.0.16 [14]. 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 QF_LIA benchmarks belong to these two sets. The benchmarks in *nec-smt* set are generated by the SMT-based BMC engine inside F-Soft [3], and the benchmarks in *rings* encode associativity properties on modulo 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.

For the evaluation of our preprocessor, we generated a set of simplified benchmarks out of the *nec-smt* benchmarks and ran the experiments 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-Ite*, and the third one is the number of *Term-Ite* after the simple preprocessing. The last column gives the rate of the reduction. On average, we achieved 15 percent of *Term-Ite* reduction with the simple preprocessing of Section 4.

To assess the effectiveness of our approach, we compared our approach with the naive approach of Eq. 3.1. As Fig. 15 shows, our approach is significantly better than the naive approach. In addition, we disabled theory simplification in the 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.

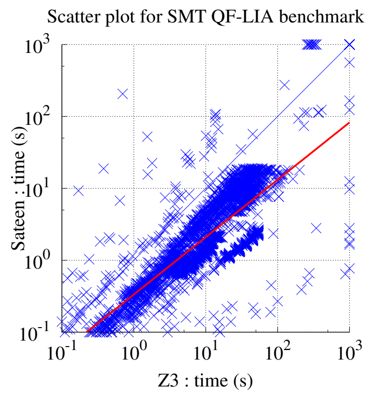

Fig. 8. Z3 vs. Sateen on QF_LIA

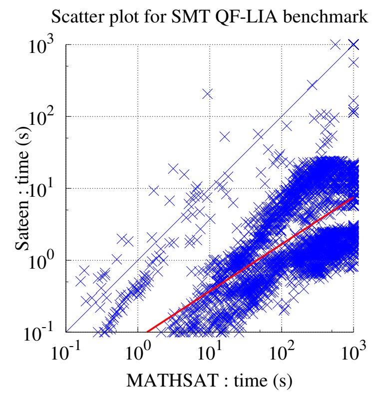

Fig. 9. MATHSAT vs. Sateen on QF_LIA

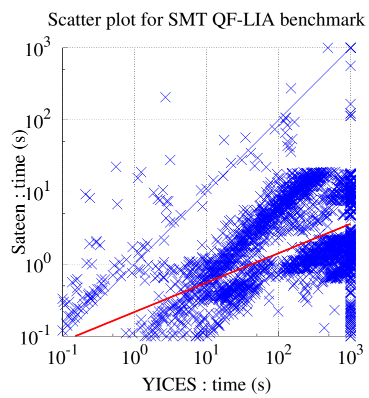

Fig. 10. YICES vs. Sateen on QF_LIA

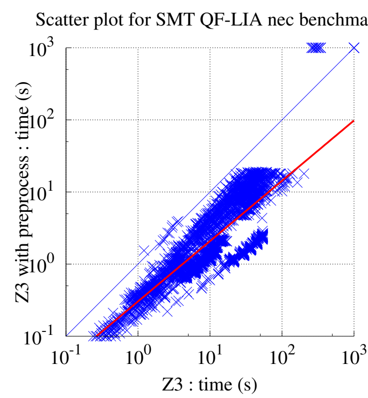

Fig. 11. Z3 WITH PREPROCESS vs. Z3 on QF_LIA

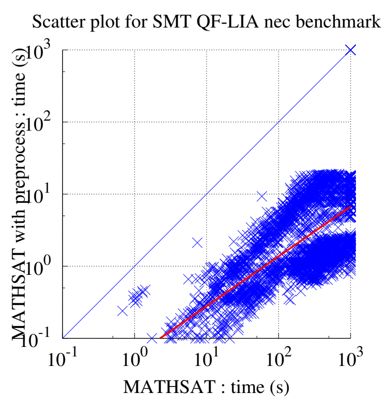

Fig. 12. MATHSAT WITH PREPROCESS vs. MATHSAT on QF_LIA

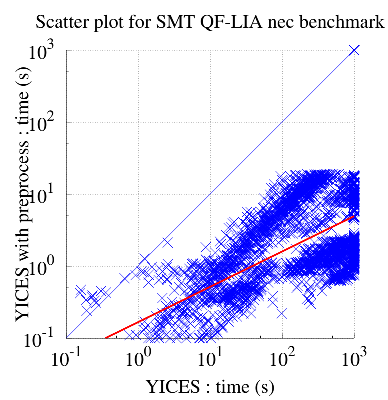

Fig. 13. YICES WITH PREPROCESS vs. YICES on QF_LIA

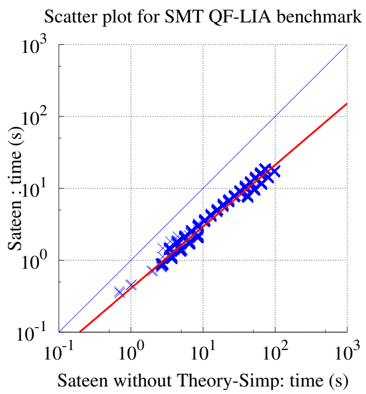

Fig. 14. SATEEN vs. Sateen without Theory-Simp on QF_LIA

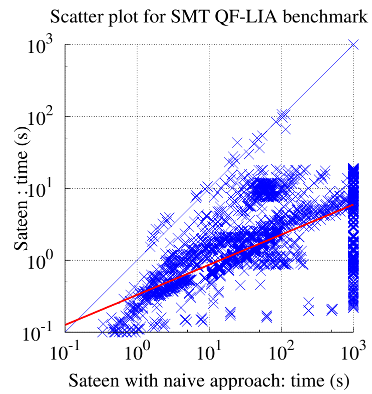

Fig. 15. SATEEN vs. Sateen with naive approach on QF_LIA

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

8 Conclusions

We have presented an algorithm for the *Term-Ite* conversion in *LA*. The approach is based on the computation of cofactors and theory simplification. The simplification is done by detecting terminal cases on the formula or using theory propagation on the atomic predicates. Experiments show that this approach is very effective in most of QF_LIA benchmarks compared to the other SMT solvers. On the other hand, since our approach may still blow up in general, it will be a good future work to find out how to combine it with the approach 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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] H. Kim and F. Somenzi. Finite instantiations for integer difference logic. In *Formal Methods in Computer Aided Design (FMCAD'06)*, pages 31–38, San Jose, CA, Nov. 2006.

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

[11] 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.

[12] 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.

[13] Url: http://smtcomp.org/.

[14] Url: http://yices.csl.sri.com.

[15] 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.