Solving Quantified Verification Conditions using Satisfiability Modulo Theories *

Yeting Ge[1], Clark Barrett[1], and Cesare Tinelli[2]

[1]New York University, yeting|barrett@cs.nyu.edu
[2]The University of Iowa, tinelli@cs.uiowa.edu

Abstract. First-order logic provides a convenient formalism for describing a wide variety of verification conditions. Two main approaches to checking such conditions are pure first-order automated theorem proving (ATP) and automated theorem proving based on satisfiability modulo theories (SMT). Traditional ATP systems are designed to handle quantifiers easily, but often have difficulty reasoning with respect to theories. SMT systems, on the other hand, have built-in support for many useful theories, but have a much more difficult time with quantifiers. One clue on how to get the best of both worlds can be found in the legacy system Simplify which combines built-in theory reasoning with quantifier instantiation heuristics. Inspired by Simplify and motivated by a desire to provide a competitive alternative to ATP systems, this paper describes a methodology for reasoning about quantifiers in SMT systems. We present the methodology in the context of the Abstract DPLL Modulo Theories framework. Besides adapting many of Simplify's techniques, we also introduce a number of new heuristics. Most important is the notion of *instantiation level* which provides an effective mechanism for prioritizing and managing the large search space inherent in quantifier instantiation techniques. These techniques have been implemented in the SMT system CVC3. Experimental results show that our methodology enables CVC3 to solve a significant number of quantified benchmarks that were not solvable with previous approaches.

1 Introduction

Many verification problems can be solved by checking formulas in first-order logic. Automated theorem proving (ATP) systems are much more powerful than those of just a few years ago. However, practical verification conditions often require reasoning with respect to well-established first-order theories such as arithmetic. Despite their power, ATP systems have been less successful in this domain. A new breed of provers, dubbed SMT solvers (for Satisfiability Modulo Theories) is attempting to fill this gap.

Solvers for SMT are typically based on decision procedures for the satisfiability of quantifier-free formulas in certain logical theories of interest. Until recently,

* This work was partially supported by a donation from Intel Corp. and by the National Science Foundation under grants 0237422 and 0551645.

they have been traditionally rather limited in their ability to reason about quantifiers, especially when compared to ATP systems. A notable exception is the prover Simplify [11] which combines a Nelson-Oppen style prover with heuristics for instantiation of quantifiers. Simplify has been successfully applied in a variety of software verification projects including ESC/Java [13], and for many years has represented the state-of-the-art in SMT reasoning with quantifiers.

However, Simplify has a number of drawbacks. Chief among them is the fact that it is old and no longer supported. Additionally, there are several weaknesses in Simplify's heuristics, so that often users must spend considerable manual effort rewriting or annotating their input formulas before Simplify can successfully prove them. Finally, modern SMT solvers have a host of performance and feature enhancements that make them more appealing for use in applications. Unfortunately, users of SMT solvers have had to choose between these improvements and effective quantifier support.

In this paper we discuss efforts to update and improve quantifier reasoning in SMT solvers based on the DPLL(T) architecture [14]. We begin by extending the Abstract DPLL Modulo Theories framework [17], a convenient abstract framework for describing such systems, with rules for quantifiers. We then explain the main heuristics employed by Simplify as strategies within this framework, and introduce several improvements to Simplify's strategies. Most novel is the notion of *instantiation level* which is an effective means of prioritizing and managing the many terms that are candidates for quantifier instantiation.

The techniques discussed in the paper have been implemented in CVC3 [4], a modern DPLL(T)-style solver based on a variant of the Nelson-Oppen combination method [5, 6]. We conclude with a set of experimental results demonstrating the effectiveness of our heuristics in improving the performance of CVC3 and in solving verification conditions (in particular, several from the NASA suite introduced in [10]) that, at the time the experiments were run, no previous ATP or SMT system had been able to solve.

2 Background

We will assume the usual notions and adopt the usual terminology for first-order logic with equality. We also assume familiarity with the fundamentals of unification theory (see, e.g., [1]). For brevity, when it is clear from context, we will refer to an atomic formula also as a term. If φ is a first-order formula or a term, t is a term, and x is a variable, we denote by $\varphi[x/t]$ the result of substituting t for all free occurrences of x in φ. That notation is extended in the obvious way to tuples $\overline{x}$ of variables and $\overline{t}$ of terms of the same length. The notation $\exists\overline{x}.\varphi$ stands as usual for a formula of the form $\exists x_1.\exists x_2.\cdots\exists x_n.\varphi$ with φ not starting with an existential quantifier (similarly for $\forall\overline{x}.\varphi$).

The *Satisfiability Modulo Theories* problem consists of determining the satisfiability of some closed first-order formula φ, a *query*, with respect to some fixed *background theory* T with signature Σ. Often it is also desirable to allow the formula to contain additional *free symbols*, i.e. constant, function, and predicate

UnitPropagate :

$$M \parallel F, C \vee l \quad \Longrightarrow \quad M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \quad \Longrightarrow \quad M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F, C \quad \Longrightarrow \quad \mathit{Fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Restart :

$$M \parallel F \quad \Longrightarrow \quad \emptyset \parallel F$$

T-Propagate :

$$M \parallel F \quad \Longrightarrow \quad M\ l \parallel F \quad \textbf{if} \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

T-Learn :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

T-Forget :

$$M \parallel F, C \quad \Longrightarrow \quad M \parallel F \quad \textbf{if} \ \{ F \models_T C$$

T-Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \quad \Longrightarrow \quad M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models_T C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Fig. 1. Transition rules of Abstract DPLL Modulo Theories. In the rules, a comma is used to separate clauses of the CNF formula, C and l respectively denote a clause and a literal, $\models$ is propositional entailment, and $\models_T$ is first-order entailment modulo the background theory T.

symbols not in Σ. We say that φ is *T-satisfiable* if there is an expansion of a model of T to the free symbols in φ that satisfies φ. Typical background theories in SMT are (combined) theories T such that the T-satisfiability of ground formulas (i.e., closed quantifier-free formulas possibly with free symbols) can be decided by a special-purpose and efficient procedure we call a *ground SMT solver*.

Most modern ground SMT solvers integrate a propositional SAT solver based on the DPLL procedure with a *theory solver* which can check satisfiability of sets of literals with respect to some fragment of T. The Abstract DPLL Modulo Theories framework [2, 17] provides a formalism for this integration that is abstract enough to be simple, yet precise enough to model many salient features of these solvers. The framework describes SMT solvers as transition systems, i.e., sets of *states* with a binary relation $\Longrightarrow$ over them, called the *transition relation*, defined declaratively by means of *transition rules*. A state is either the distinguished state *Fail* (denoting T-unsatisfiability) or a pair of the form $M \parallel F$, where F is a formula in conjunctive normal form (CNF) being checked for satisfiability and M is a sequence of literals forming a partial assignment for the literals in F. A literal l in M may contain the special annotation l^{d} indicating that it is a decision literal.

Assuming an initial state of the form $\emptyset \parallel F_0$, with $\emptyset$ denoting the empty sequence of literals, the goal of the transition rules is to make progress towards a *final* state while maintaining equisatisfiability of the formula F_0. A final state is either *Fail* or a state $M \parallel F$ such that (i) the set of literals in M is T-satisfiable, and (ii) every clause in F is propositionally satisfied by the assignment induced by M (i.e., assuming that the literals in M are all true). In the latter case, the original formula F_0 is T-satisfiable.

The transition rules are shown in Figure 1. We refer the reader to [2, 17] for a complete description of the framework.

3 Reasoning with Quantifiers in SMT

While many successful ground SMT solvers have been built for a variety of theories and combinations of theories, extending SMT techniques to quantified queries has proven so far quite difficult. This mirrors the difficulties encountered in first-order theorem proving, where quantified queries are the norm, in embedding background theories *efficiently* into existing refutation-based calculi.

Following Stickel's original idea of *theory resolution* [21], several first-order calculi have been given sound and complete theory extensions that rely on the computation of complete sets of theory unifiers. These nice theoretical results have, however, failed to generate efficient implementations thus far, mostly due to the practical difficulty, or the theoretical impossibility, of computing theory unifiers for concrete background theories of interest.

Recently, attempts have been made to embed ground SMT procedures into successful first-order provers, most notably Vampire [20] and SPASS [24], while aiming at practical usefulness as opposed to theoretical completeness (see, e.g., [19]). The work described here follows the alternative, also incomplete, approach of extending SMT solvers with effective heuristics for quantifier instantiation.

3.1 Modeling Quantifier Instantiation

The Abstract DPLL Modulo Theories framework can be easily extended to include rules for quantifier instantiation. The key idea is to allow also closed quantified formulas wherever atomic formulas are allowed. We define an *abstract atomic formula* as either an atomic formula or a closed formula of the form $\forall \bar{x} \varphi$ or $\exists \bar{x} \varphi$. An *abstract literal* is either an abstract atomic formula or its negation; an abstract clause is a disjunction of abstract literals. Then, we simply replace

$\exists$-Inst :

$$M \parallel F \implies M \parallel F,\, \neg\exists\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{c}] \quad \textbf{if} \begin{cases} \exists\overline{x}.\varphi \text{ is in } M \\ \overline{c} \text{ are fresh constants} \end{cases}$$

$\forall$-Inst :

$$M \parallel F \implies M \parallel F,\, \neg\forall\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{s}] \quad \textbf{if} \begin{cases} \forall\overline{x}.\varphi \text{ is in } M \\ \overline{s} \text{ are ground terms} \end{cases}$$

Fig. 2. New transition rules.

ground literals and clauses with their abstract counterparts. For instance, non-fail states become pairs $M \parallel F$ where M is a sequence of abstract literals and F is a conjunction of abstract clauses.

With this slight modification, we can add the two rules in Figure 2 to Abstract DPLL to model quantifier instantiation. For simplicity and without loss of generality, we assume here that abstract literals in M appear only positively (if they are negated, the negation can be pushed inside the quantifier) and that the bodies of abstract atoms are themselves in abstract CNF.

The $\exists$-Inst rule identifies a quantified abstract literal $\exists\overline{x}.\varphi$ currently in M. This formula is then skolemized by introducing fresh constants $\overline{c}$ to get $\varphi[\overline{x}/\overline{c}]$. A clause is then added that is equivalent to the implication $\exists\overline{x}.\varphi \rightarrow \varphi[\overline{x}/\overline{c}]$. Note that we cannot just add $\varphi[\overline{x}/\overline{c}]$ because the Abstract DPLL Modulo Theories framework requires that the satisfiability of F be preserved by every rule. While $\varphi[\overline{x}/\overline{c}]$ is equisatisfiable with $\exists\overline{x}.\varphi$, the latter is an assumption that depends on M, so in order to make F independent of M, the assumption must be included in the added clause.

The $\forall$-Inst rule works analogously except that the formula is *instantiated* instead of skolemized. Any ground terms can be used for the instantiation.

Example 1. Suppose a and b are free constant symbols and f is a unary free function symbol. We show how to prove the validity of the formula $(0 \leq b \wedge (\forall\, x.\, x \geq 0 \rightarrow f(x) = a)) \rightarrow f(b) = a$ in the union T of rational arithmetic, say, and the empty theory over $\{a, b, f\}$. We first negate the formula and put it into abstract CNF. Three abstract unit clauses are the result:

$$0 \leq b \quad \wedge \quad \forall\, x.\, (x \not\geq 0 \vee f(x) = a) \quad \wedge \quad f(b) \neq a\ .$$

Let l_1, l_2, l_3 denote the three abstract literals in the above clauses. Then the following is a derivation in the extended framework:

$$\begin{array}{lrcll} & \emptyset & \parallel & l_1,\, l_2,\, l_3 & \text{(initial state)} \\ \Longrightarrow^* & l_1\; l_2\; l_3 & \parallel & l_1,\, l_2,\, l_3 & \text{(by UnitPropagate)} \\ \Longrightarrow & l_1\; l_2\; l_3 & \parallel & l_1,\, l_2,\, l_3,\, \neg l_2 \vee b \not\geq 0 \vee f(b) = a & \text{(by } \forall\text{-Inst)} \\ \Longrightarrow & l_1\; l_2\; l_3\; b \geq 0 & \parallel & l_1,\, l_2,\, l_3,\, \neg l_2 \vee b \not\geq 0 \vee f(b) = a & \text{(by } T\text{-Propagate)} \\ \Longrightarrow & \mathit{Fail} & & & \text{(by Fail)} \end{array}$$

The last transition is possible because M falsifies the last clause in F and contains no decisions (case-splits). As a result, we may conclude that the original clause set is T-unsatisfiable, which implies that the original formula is valid in T.

It is not hard to see, using an analysis similar to that in [2], that the $\exists$-Inst and $\forall$-Inst rules preserve the satisfiability of F and therefore the soundness of the transition system. It is also clear that termination can only be guaranteed by limiting the number of times the rules are applied. Of course, for a given existentially quantified formula, there is no benefit to applying $\exists$-Inst more than once. On the other hand, a universally quantified formula may need to be instantiated with several different ground terms to discover that a query is unsatisfiable. For some background theories (e.g., universal theories), completeness can be shown for exhaustive and fair instantiation strategies that consider all possible quantifier instantiations by ground terms. This result, however, is of little practical relevance because of the great inefficiency of such a process.

In this paper we do not focus on finding a complete strategy. Rather, we focus on strategies for applying $\forall$-Inst that acheive good *accuracy*, understood ideally here as the ratio of proved over unproved unsatisfiable queries in a given set. Then we discuss and evaluate their implementation within our CVC3 solver.

Also, though we do not discuss it further in this paper, another important feature of these rules is that they fit in nicely with CVC3's proof mechanism. CVC3 has an option to produce a proof object when it determines that a formula is unsatisfiable. We have added the quantifier proof rules described above to CVC3, so that it is now able to produce proofs also for refuted formulas containing quantifiers.

4 Strategies for Instantiation

4.1 Instantiation via Matching

A naive strategy for applying rule $\forall$-Inst is the following: once $\forall$-Inst has been selected for application to an abstract literal $\forall\,\overline{x}.\varphi$, the rule is repeatedly applied until $\overline{x}$ has been instantiated with every possible tuple of elements from some finite set G of ground terms. A reasonable choice for G is the set of ground terms that occur in assumed formulas (i.e., in M). We call this approach *naive instantiation*. A refinement of this strategy for sorted logics is to instantiate $\overline{x}$ with all and only the ground tuples of G that have the same sort as $\overline{x}$.

Naive instantiation is sufficient for solving benchmarks for which the number of possible instantiations is small. However, there are many verification conditions for which naive instantiation is hopelessly inefficient because of the large number of candidates for instantiation.

The Simplify prover uses a better heuristic, that still applies $\forall$-Inst exhaustively to an abstract atom, but selects for instantiation only ground terms that are *relevant* to the quantified formula in question, according to some heuristic relevance criteria. In terms of Abstract DPLL Modulo Theories, Simplify's main relevance criterion can be explained as follows: given a state $M \parallel F$ and an abstract literal $\forall\overline{x}.\varphi$ in M, try to find a non-variable sub-term t of $\forall\overline{x}.\varphi$ containing the variables $\overline{x}$, a ground term g in M, and ground terms $\overline{s}$, such that $t[\overline{x}/\overline{s}]$ is equivalent to g modulo E, written $t[\overline{x}/\overline{s}] =_E g$, for a certain set E of equalities entailed by $M \cup T$.[1] In this case, we expect that instantiating $\overline{x}$ with $\overline{s}$ is more likely to be helpful than instantiating with other candidate terms. Following Simplify, we call the term t a *trigger* (for $\forall \overline{x}.\varphi$).

Example 2. Consider again the formula in Example 1. At the point where ∀-Inst is applied, M consists of the following sequence of literals:

$$0 \leq b, \quad \forall\, x.\,(x \ngeq 0 \vee f(x) = a), \quad f(b) \neq a\ .$$

There are four ground terms appearing in M: 0, a, b, and $f(b)$. Thus, naive instantiation would apply ∀-Inst four times, once for each ground term. On the other hand, Simplify's matching heuristic would first identify a trigger in $\forall\, x.(x \ngeq 0 \vee f(x) = a)$. Since a trigger must be a term properly containing the quantified variable, the only candidate is $f(x)$. Now the trigger is compared with the set of ground terms. There is a single match, with $f(b)$, obtained when x is bound to b. Thus, the matching heuristic selects the ground term b for instantiation.

In unification theory parlance, Simplify finds ground instantiations of universally quantified variables by solving *E-matching* problems of the form $t =^? g$ where t is a trigger and g is a ground term in M. Ideally, one would like E to be the set $E_{M \cup T}$ of all the equalities entailed by $M \cup T$. In general, this is not feasible because of theoretical or practical limitations depending on the specific background theory T. Hence, the need to consider suitable subsets of $M \cup T$. In Simplify's case E consists of the congruence closure $CC(M)$ of the ground equalities belonging to M (see [11] for details). We adopt the same restriction in this work as well, and use an E-matching algorithm similar to the one used by Simplify.

Some classes of equational theories E, such as theories of an associative, commutative and idempotent (ACI) symbol or the theory of Abelian groups (AG), have both a decidable E-matching problem and efficient procedures for computing complete sets of E-matchers (see [1]). When the background theory T contains AG, say, one could then imagine combining an algorithm for AG-matching with one for $CC(M)$-matching, with the goal of increasing the number of relevant matchers for each trigger. The investigation of this possibility is however left to future work.

4.2 Eager Instantiation versus Lazy Instantiation

So far, we have been concerned with the question of *how* to apply the rule ∀-Inst to a given abstract atom. An orthogonal question is *when* to apply it. One strategy, which we call *lazy instantiation*, is to apply ∀-Inst only when it is the only applicable rule. At the opposite end of the spectrum, another strategy, which we call *eager instantiation*, is to apply ∀-Inst to a universally quantified formula as soon as possible (i.e., as soon as it is added to the current M).

[1] That is, equalities $t_1 = t_2$ such that $M \models_T t_1 = t_2$.

In Simplify, propositional search and quantifier instantiation are interleaved. When Simplify has a choice between instantiation and case splitting, it will generally favor instantiation. Thus, Simplify can be seen as employing a form of eager instantiation.[2] Others [12] have advocated the lazy approach. One advantage of lazy instantiation is that an off-the-shelf SAT solver can be used. Eager instantiation typically requires a more sophisticated SAT solver that can accept new variables and clauses on the fly. On the other hand, this additional functionality is already required even in the quantifier-free case in order to enable important efficiency gains (see [17], for example). For SMT systems that use these techniques, CVC3 included, eager instantiation does not require significant additional infrastructure.

We compare eager and lazy instantiation in Section 6 below.

5 Improving Instantiation Strategies

In this section we describe several improvements to the basic strategies discussed above. These strategies are implemented in CVC3 and evaluated in Section 6.

5.1 Triggers

Consider a generic quantified formula $\forall \overline{x}.\varphi$. The first step in the matching strategy described above is to find triggers within φ. CVC3 improves on Simplify's automated trigger generation methods in several ways.

In CVC3, every sub-term or non-equational atom t of φ that contains all the variables in $\overline{x}$ and at least one function or predicate symbol is considered a viable trigger. For example, if $\overline{x} = (x_1, x_2)$, then $x_1 \leq x_2$ and $g(f(x_1 + y), x_2)$ are legal triggers, but $0 \leq x_1$ and $f(x_1) + 1 = x_2$ are not.

Simplify is slightly more restrictive: it requires that a trigger contain no additional variables besides those in $\overline{x}$. For example, in the formula $\forall x.(f(x) \rightarrow \forall y.g(x,y) < 0)$, the term $g(x,y)$ is not a viable trigger for Simplify because it contains y which is not bound by the outermost quantifier. Our experiments indicate that this restriction may be unnecessary. More specifically, it does cause a loss of accuracy in some cases (in particular, CVC3's better performance on the *nasa* benchmarks described in Section 6.3 is partly due to relaxing this restriction).

Avoiding instantiation loops Simplify uses a simple syntactic check to prevent its instantiation mechanism from diverging; specifically, it discards a potential trigger t if certain (syntactical) instances of t occur elsewhere in the formula. For example, in $\forall x.P(f(x), f(g(x)))$, the term $f(x)$ will not be selected as a trigger because an instance of $f(x)$, namely $f(g(x))$ occurs in the formula.

[2] Simplify's choice of eager instantiation is especially appropriate for Extended Static Checking (the application for which it was developed), where a compiler provides *patterns* telling Simplify which triggers to use, helping to avoid too many instantiations.

While simple and inexpensive, this static filtering criterion is unable to detect more subtle forms of loops, as shown in the following example.

Example 3. Consider a state $M \parallel F$ with M containing the abstract literal $\psi = \forall x.\, (x > 0 \rightarrow \exists y.\, f(x) = f(y) + 1)$ where f is free. The only trigger for ψ is $f(x)$ and Simplify has no reason to reject this trigger.

Now, if the set of ground terms contains $f(3)$, say, then with an application of $\forall$-Inst, it is possible to add the abstract clause $\neg\psi \vee \exists y.\, f(3) = f(y)+1$ to F. Then, with an application of UnitPropagate and of $\exists$-Inst the literal $f(3) = f(c_1) + 1$, with c_1 fresh, can be added to M. The introduction of $f(c_1)$ in the set of ground terms can now give rise to a similar round of rule applications generating a new term $f(c_2)$, and so on.

To prevent instantiation loops like those in the example above, in addition to Simplify's static loop detection method, CVC3 also implements a general method for dynamically recognizing loops (including loops caused by groups of formulas together) and disabling the offending triggers. However, we do not describe that method here because the *instantiation level* heuristic described in Section 5.5 below is much more effective.

Multi-trigger generation Sometimes, there are no triggers that contain all the variables in $\overline{x}$. In this case, Simplify generates a *multi-trigger*: a small set of terms in φ which together contain all (and exactly) the free variables in $\overline{x}$.[3]

CVC3 has essentially the same mechanism, but it includes a heuristic for selecting which multi-trigger to choose. It does this by putting together in a multi-trigger only atoms having the same *polarity*.[4] For example, given the abstract CNF formula

$$\forall x, y, z.(\neg P(x, y) \vee \neg Q(y, z) \vee R(x, z)) \ ,$$

CVC3 will choose the set $\{P(x, y), Q(y, z)\}$ as a multi-trigger because $P(x, y)$ and $Q(y, z)$ have the same polarity (negative) and together contain all bound variables, but will not choose the set $\{P(x, y), R(y, z)\}$ because $R(y, z)$ has positive polarity. This heuristic is especially helpful for directing the instantiation of axioms specifying transitivity or antisymmetry of binary predicates.

[3] In fact, sometimes a multi-trigger can result in a more complete procedure, even if a trigger exists containing all of the variables. For instance, in the axiom $\forall\, A\, i\, j\, v.\, (i = j \rightarrow \mathit{read}(\mathit{write}(A, i, v), j) = \mathit{read}(A, j))$, the term $\mathit{read}(\mathit{write}(A, i, v), j)$ is a trigger, but the multi-trigger $\{\mathit{write}(A, i, v), \mathit{read}(A, j)\}$ can refute formulas that cannot be refuted using the single trigger. The authors thank one of the anonymous reviewers for bringing this example to our attention. CVC3 does not use multi-triggers unless there are no standard triggers containing all the variables.

[4] Given an abstract CNF formula F, the polarity of an atom in F is positive if the atom appears only positively in F, negative if the atom occurs only negated in F, and both if the atom occurs both positively and negatively.

5.2 Matching algorithm

As we mentioned in the previous section, when looking for E-matches for triggers, CVC3 chooses as the equational theory E a rather restricted subset of the equalities entailed by $M \cup T$ where T is the built-in background theory and M is the current set of assumed abstract literals. The main motivation for the restriction is that the corresponding E-matching algorithm is easier to implement efficiently and, as we show in the next section, provides good results experimentally. A high-level description of the algorithm is provided below.

As M is modified, CVC3 also computes and stores in its data structures the congruence closure E of the (positive) ground equations in M over the set G of all ground terms in M. For any theory T, any two terms equal modulo E are also equal modulo $T \cup M$. Then, to apply the rule $\forall$-Inst to an abstract literal $\forall \overline{x}.\varphi$, CVC3 generates ground instantiations for $\overline{x}$ by E-matching the triggers of $\forall \overline{x}.\varphi$ against all the terms in G. CVC3 implements a sound and terminating E-matching procedure by extending the standard rule-based syntactic unification algorithm as explained below.

Given a trigger t of the form $f(t_1, \ldots, t_n)$ where f is a free symbol, we select from G all terms of the form $f(s_1, \ldots, s_n)$; for each of these terms we then try to solve the (simultaneous) unification problem $\{t_1 =^? s_1, \ldots, t_n =^? s_n\}$. Note that this is in effect a matching problem because all the right-hand sides $s_1, \ldots, s_n$ are variable-free terms. Standard unification fails when it encounters the case $g(\overline{t}) =^? g'(\overline{s})$ where g and g' are distinct symbols. In contrast, we do not immediately fail in this case.

In general, when we select an equation of the form $g(\overline{t}) =^? s$ with s ground, we do not fail in the following two sub-cases: (i) $g(\overline{t})$ is ground and $g(\overline{t}) =_E s$,[5] and (ii) g is a free symbol and there is a term of the form $g(\overline{u})$ in G such that $s =_E g(\overline{u})$. In the first case, we just remove the equation $g(\overline{t}) =^? s$; in the second case, we replace it by the set of equations $\overline{t} =^? \overline{u}$.

For a simple example, consider matching a trigger like $f(h(x))$ with a ground term $f(a)$ where f, h, a are free symbols and x is a variable. Suppose that $a = h(s) \in E$ for some s. Then the procedure above can generate the non-syntactic unifier $\{x \mapsto s\}$.

It is not difficult to see using standard soundness and termination arguments that this unification procedure converges, and when it does not fail it produces a grounding E-unifier (in fact, an E-matcher) for the problem $f(t_1, \ldots, t_n) =^? f(s_1, \ldots, s_n)$. This unifier is applied to the body of the abstract atom $\forall \overline{x}.\varphi$ to obtain the clause for $\forall$-Inst.

Implementation The implementation of the CVC3 E-matching algorithm is given in more detail in the pseudo-code in Figure 3.[6] The algorithm makes use of

[5] Due to the way the congruence closure E is maintained in CVC3, checking that $g(\overline{t}) =_E s$ takes nearly always constant time.

[6] We use imperative pseudo-code description instead of the more abstract level of description used elsewhere in the paper to provide a view of the algorithm that is closer to the actual implementation.

an abstract data type for bindings. A binding is a partial function from variables to ground terms. If v is a variable, we denote the result of applying a binding *binding* to v as *binding*$[v]$ and we say that v is *bound* in *binding* if *binding*(v) exists.

The main matching function is `recMultMatch`. It takes three parameters: a ground term *gterm*, a trigger *vterm*, and a binding *binding*. The goal of the function is to match *gterm* with *vterm* in a way that is consistent with the given binding *binding*. It returns a set of bindings that achieve this goal (this set may be empty). The first case considered is when *vterm* is variable. If *vterm* is bound in *binding*, then matching can only succeed in this case if *binding*[*vterm*] and *gterm* are equivalent modulo E. Otherwise, the binding is extended by binding *vterm* to *gterm*. The case when *vterm* is ground is similar to when *vterm* is bound. There is a match if *vterm* is equivalent to *gterm* modulo E.

If *vterm* is neither a variable nor ground, then it must be a function application. In this case, we need to match the function symbol as well as all the children. We first make use of an auxiliary function, `equivalent`. The function `equivalent(`*gterm*`,` *vterm*`)` returns all terms that are equivalent modulo E to *gterm* and begin with the same symbol as *vterm*. For example, if E contains $a = f(b)$ and $a = g(c)$, then `equivalent(`a`,` $g(d)$`)` includes $g(c)$. Once we have the set of ground terms starting with the same symbol as *vterm*, we go through the set one by one and try matching the children.

The children are matched using the function `multMatchChild`. This function takes a ground term *gterm*, trigger *vterm*, and binding *binding* as above, except there is a precondition that *gterm* and *vterm* must begin with the same function symbol. The function iterates through each child building up a set of bindings. The bindings returned from the result of matching child i are each considered as possible bindings for matching child $i + 1$. The set of all bindings found are returned.

5.3 Heuristics and Optimizations.

The instantiation mechanism above applies to triggers whose top symbol is a free function symbol. Triggers whose top symbol is a theory symbol are currently treated the same way unless the symbol is an arithmetic symbol. Triggers starting with $+$ or $*$ are just discarded because treating those symbols syntactically is ineffectual and treating them semantically, as AC symbols say, is too onerous. A trigger t of the form $t_1 < t_2$ or $t_1 \leq t_2$ is considered as long as t_1 and t_2 are not both variables (that would generate too many matches) and is processed as follows.[7] For every ground atom p of the form $s_1 < s_2$ or $s_1 \leq s_2$ in M, CVC3 generates the E-matching problem $\{t_1 =^? s_2, t_2 =^? s_1\}$ if t has positive polarity and p occurs in M, or t has negative polarity and $\neg p$ occurs in M; otherwise it generates the problem $\{t_1 =^? s_1, t_2 =^? s_2\}$. The rationale of this heuristic is best explained with an example. Suppose M contains the following abstract literals:

[7] In CVC3, atoms using $>$ and $\geq$ are normalized internally to $<$ and $\leq$ atoms, respectively.

```
recMultMatch(gterm, vterm, binding) {
 if (vterm is a variable) {
   if (vterm is bound in binding) {
     if (gterm =_E binding[vterm]) {
       return { binding };
     }
     else {
       return ∅;
     }
   }
   else {
     binding[vterm] := gterm;
     return { binding };
   }
 }
 else if (vterm is ground) {
   if (gterm =_E vterm) {
     return { binding };
   }
   else {
     return ∅;
   }
 }
 else {
   allGterms := equivalent(gterm, vterm);
   newBindings := ∅;
   foreach (g in allGterms) {
     newBindings := newBindings ∪ multMatchChild(g, vterm, binding);
   }
   return newBindings;
 }
}


multMatchChild(gterm, vterm, binding){
 newBindings := { binding };
 for (i := 0; i < gterm.arity(); i++) {
   nextBindings := ∅;
   foreach (binding in newBindings) {
     nextBindings := nextBindings ∪
                     recMultMatch(gterm[i], vterm[i], binding);
   }
   newBindings := nextBindings;
 }
 return newBindings;
}
```

Fig. 3. Matching algorithm

1. $\forall x, y. (\neg x < y \vee f(x) < f(y))$
2. $a < b$
3. $f(b) < f(a)$

If we directly match $f(b) < f(a)$ with $f(x) < f(y)$, we will get $b < a \rightarrow f(b) < f(a)$. At this point, no further matching is possible and no contradiction can be deduced. If, however, we instead match $f(a) < f(b)$ with $f(x) < f(y)$, we will derive $a < b \rightarrow f(a) < f(b)$. Then, from $(a < b)$, $(a < b \rightarrow f(a) < f(b))$, and $f(b) < f(a)$, a contradiction can be deduced. The intuition is that $f(b) < f(a)$ implies $\neg(f(a) < f(b))$ and contradictions are deduced when matching atoms of opposite polarity.

When used within the DPLL(T) architecture, the matching algorithm must be invoked many times. Given n ground terms and m triggers, a naive approach is to do matching for all mn pairs of ground terms and triggers. CVC3 improves on this approach as follows. At all times, a map is maintained that maps each function symbol to the list of triggers beginning with that symbol. When a ground term a with top symbol f is to be matched, we fetch the whole list of triggers with top symbol f and then match a with every trigger in the list. More sophisticated techniques for matching multiple triggers at the same time are described in [9].

A trigger is *simple* if all its proper sub-terms are variables. When matching a ground term with a simple trigger, say $f(x, y)$, as long as the ground term's top symbol is f, the matching is always successful. CVC3 keeps track of simple triggers and avoids calling the matching algorithm on them.

Two triggers are *α-equivalent* if they can be rewritten into each other using a bijective renaming of the variables. For example, $f(x, a)$ and $f(y, a)$ are α-equivalent, where a is a ground term and x and y are variables. CVC3 detects triggers that are α-equivalent and only matches against a single representative trigger for each such set of triggers.[8]

Two ground terms may produce redundant results as well. In particular if E contains $a = g(b)$, then the ground terms $f(a)$ and $f(g(b))$ will produce the same result. CVC3 maintains a unique equivalence class representative for each equivalence class induced by E. We define the *signature* of a term to be the result of replacing each of its children with its equivalence class representative. CVC3 avoids redundancy by tracking sets of ground terms with the same signature and only using one of the ground terms from each such set during matching.

A final important consideration is the interaction of matching with the DPLL search. CVC3 keeps track of which terms and triggers have been matched along the current branch of the search tree so that these matches are not repeated. However, if a new equality is asserted along a branch, it may be the case that a pair which previously was attempted and did not match now does match as a consequence of the new equality. For example, suppose ground term $f(a)$ is matched with $f(g(b))$. If a is not known to be equivalent to $g(b)$, the matching algorithm fails. However, suppose that later along the same branch, $a = g(b)$ is

[8] This is equivalent to using term indexing for bound variables, as is common in many ATP systems.

asserted. Then we should try to match $f(a)$ and $f(g(b))$ again because the new equality enables a match.

One approach for handling this problem is to use the *inverted path tree* data structure introduced in [9]. CVC3 employs a simple alternative solution. The basic idea is to periodically retry pairs of ground terms and triggers that did not match before. Since this operation is expensive, CVC3 does it lazily, that is only when all other heuristics fail to produce a contradiction. In addition, CVC3 only considers pairs involving a ground term for which one or more of its proper sub-terms appears in an equivalence class that has changed. This acheives a similar effect as Simplify's *mod-time* heuristic [11]. Notice also that simple triggers never need to be retried.

5.4 Special Instantiation Heuristics

In addition to E-matching, CVC3 also employs some specialized instantiation heuristics that have proven useful on the kinds of formulas that appear in practical verification conditions. For simplicity, we will refer to these heuristics too as "trigger matching" even if they are not based on matching in the technical sense of unification theory.

As mentioned above, the polarity heuristic is useful for recognizing axioms specifying that a certain free predicate symbol is required to be antisymmetric or transitive.

Another heuristic applies to formulas involving CVC3's built-in theory of arrays, which defines a read and a write operator. All triggers of the form $\mathit{read}(\mathit{write}(a, x, v), i)$ where x is one of the quantified variables, in addition to acting as normal triggers, also cause x to be instantiated to the index term j of any ground term of the form $\mathit{read}(a, j)$ or $\mathit{write}(a, j, u)$. The rationale is that when instantiating a variable that is used as an index to an array, we want to consider all known ground array index terms. Usually, there are not too many of these terms but the standard matching techniques do not discover all of them.

5.5 Trigger Matching by Instantiation Levels

In SMT problems coming from verification applications, one of the main targets of CVC3, the query is a formula of the form $\Gamma \wedge \neg\varphi$ where φ is a verification condition and Γ is a large and more or less fixed T-satisfiable collection of (quantified) axioms about a number of relations and functions that are relevant to the verification application but for which there is no built-in solver. A large number of these axioms typically have no bearing on whether the negation of a particular verification condition is T-satisfiable with Γ. With heuristic instantiation, this entails that too many resources might be easily spent in producing and processing instances of axioms unrelated to the formula φ.

Simplify uses a *matching depth* heuristic to try to address this problem. Each time a new clause is generated by quantifier instantiation, it is assigned a numerical value which is one greater than the largest value assigned so far. This value is the matching depth of the clause. Later, when a literal must be chosen for a case-split, literals from clauses with a lower matching depth are preferred to those with a higher matching depth. A limit on matching depth is also used to determine when to give up and terminate.

To acheive these same goals, CVC3 uses a different approach, better suited to systems with a DPLL(T) architecture—where case splitting is not necessarily clause-based. Instead of giving a score to clauses, CVC3 assigns an *instantiation level* to every ground term it creates. Intuitively, an instantiation level n for a term t indicates that t is the result of n rounds of instantiation. More precisely, all terms in the original query are given an instantiation level of 0. If a formula $\forall\, x.\, \varphi$ is instantiated with the ground term s, and n is the instantiation level of t, then all the new terms in $\varphi[x/t]$ (as well as any new terms derived from them via theory reasoning) are given the instantiation level $n + 1$.

CVC3 provides as an option a trigger matching strategy that visits ground terms by instantiation levels. With this strategy, CVC3 matches triggers only against ground terms whose instantiation level is within a current upper bound b. This bound, whose initial value can be set by the user, is increased, by one, only when CVC3 reaches a (non-fail) state $M \parallel F$ where $\forall$-Inst is the only applicable rule and all terms with instantiation level less than or equal to b have already been considered.

Trigger matching by instantiation levels has proved very effective in our experiments, discussed in the next session. Here we point out that its inherent fairness has also the derived benefit of neutralizing the possible harmful effects of instantiation loops in the eager instantiation strategy. The reason is simply that each of the new ground terms generated within an instantiation level belongs by construction to the next level, and so will not be considered for matching until all other terms in the current level have been considered. As a consequence, checking for instantiation loops, either statically or dynamically, is completely unnecessary. Moreover, using instantiation levels allows us to enable by default those triggers that static or dynamic loop detection would have disabled. Significantly, we discovered that such triggers are actually necessary to prove many examples.

We observe that theorem provers for instantiation-based first-order calculi [18, 15, 7, e.g.] also use fair variable instantiation strategies—needed to guarantee refutational completeness. Many of these provers achieve fairness by instantiating variables only by terms whose *depth* is below a progressively larger bound, where a term's depth is measured as the depth of the term's abstract syntax tree, or in some other equivalent way. While simpler to implement than our instantiation level strategy, an instantiation strategy based on term depth is not suitable in our case because it does not guarantee fairness. The main problem is that CVC3 may generate an unbounded number of Skolem constants as a result of applying the $\exists$-Inst rule to new formulas generated by the $\forall$-Inst rule.[9] This means in particular that it may generate an unbounded number of ground terms of the

[9] In instantiation-based provers the number of Skolem symbols is constant during a derivation because existential quantifiers are Skolemized once and for all during a preprocessing step.

same depth. To see that, it is enough to look at Example 3 again. Limiting instantiations by term depth will not break the instantiation loop described there.

6 Experimental Results

We implemented the quantifier instantiation heuristics described here in CVC3 version 1.1. We evaluated the performance of these heuristics both within CVC3 and in comparison with other theorem provers and SMT solvers. We considered two leading automated theorem provers for first-order logic: Vampire 8.1 and SPASS 2.2, and three SMT solvers supporting quantified formulas: Simplify, yices 1.0, and Fx7[10]. This section reports on the results of our evaluation. A more detailed version of all the results discussed here can be found at `http://www.cs.nyu.edu/~barrett/cade07`.[11]

All tests were run on AMD Opteron-based (64 bit) systems, running Linux, with a timeout of 5 minutes (unless otherwise stated) and a memory limit of 1 GB.

6.1 Benchmarks

The benchmarks for our evaluation are from the SMT-LIB benchmark library [3]. Our test set consists of 29,004 benchmarks from three different SMT-LIB *logics*: AUFLIA, which is based on a theory of arrays[12], uninterpreted functions, and linear integer arithmetic; AUFLIRA, based on a theory of arrays, uninterpreted functions, and mixed linear integer and real arithmetic; and AUFNIRA based on a theory of arrays, uninterpreted functions, and mixed non-linear integer and real arithmetic. Benchmarks for these logics are further subdivided into families. In AUFLIA, there are five families: *Burns*, *misc* (we lumped the single benchmark in the *check* family in with *misc*), *piVC*, *RicartAgrawala*, and *simplify*. In AUFLIRA, there are two families: *misc* and *nasa*. And in AUFNIRA, there is a single family: *nasa*.

We will comment more specifically on two of these families, *nasa* and *simplify*, below. For more information on the other benchmarks and on the SMT-LIB library, we refer the reader to the SMT-LIB website: `http://www.smtlib.org`.

The *nasa* families make up the vast majority of the benchmarks with a total of 28,065 benchmarks in two families. These cases are *safety obligations* automatically generated from annotated programs at NASA. Following their introduction in [10], these benchmarks were made publicly available in TPTP

[10] The version of Fx7 was the one available at `http://nemerle.org/~malekith/smt/en.html` as of February 2007.

[11] These results represent the state-of-the-art as of May 2007. Partial results comparing more recent versions of some solvers can be found on the SMT competition website: `http://www.smtcomp.org`.

[12] It should be remarked that most of the benchmarks in AUFLIA make little or no use of the array theory.

format [23], a format for pure first-order logic. We then undertook the task of translating them into the SMT-LIB format and contributing them to the SMT-LIB library.

To adapt these benchmarks to SMT, several steps were required. First, we removed quantified assumptions that were determined to be valid with respect to the background theories,[13] in this case arrays and arithmetic, and made sure to use the built-in symbols defined in the SMT-LIB standard. Second, since SMT-LIB uses a many-sorted logic, we had to infer sorts for every symbol. We used the following rules to infer them:

1. The index of an array is of integer sort;
2. The return sort of the functions *cos*, *sin*, *log*, *sqrt* is real;
3. The terms on both sides of infix predicates $=$, $<=$, $>=$, $<$ and $>$, must have the same sort;
4. If the sort of a term cannot be deduced by the above rules, it is assumed to be real.

According to [10], of the 28,065 cases, only 14 are supposed to be satisfiable (the rest are unsatisfiable). However, after running our experiments and carefully examining the benchmarks in their present form in the TPTP library, our best guess is that somewhere around 150 of the cases are actually satisfiable (both in the SMT-LIB format and in the original TPTP format). It is difficult to know for sure since for these cases, no tool we are aware of can reliably distinguish between a truly satisfiable formula and one that is simply too difficult to prove unsatisfiable, and determining this by hand is extremely tedious and error-prone. We suspect that some assumptions present in the benchmarks from [10] were lost somehow before their submission to the TPTP library, but we do not know how this happened. In any case, most of the benchmarks are definitely unsatisfiable and while many are easy, a few of them are very challenging.

The other major family is the *simplify* family, which was translated (by others) from a set of over 2,200 benchmarks introduced in [11] and distributed with the Simplify theorem prover. Only a selection of the original benchmarks were translated. According to the translator, he excluded benchmarks that were too easy or involved non-linear arithmetic [8]. There are 833 benchmarks in this family, all of which are unsatisfiable.

6.2 Evaluating the Heuristics

We began by running CVC3 using only naive instantiation (trying both the lazy and eager strategies) on all SMT-LIB benchmarks. Of 29,004 benchmarks, 23,389 can be solved in negligible time by both the eager and the lazy naive strategies.[14] As a result, these benchmarks are not helpful for evaluating our

[13] These are assumptions that were added by hand to enable better performance by ATP systems. They were removed by using CVC3 to automatically check for validity. Note that this returns the benchmarks to a state more faithfully representing the original application.

[14] In fact, almost all of these can be solved without any quantifier reasoning at all. Obviously, these are not good benchmarks for testing instantiation strategies.

Lazy strategy		(i) BTBM		(ii) BTSM		(iii) STSM		(iv) IL	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	**12**	0.013	12	0.013	12	0.014	12	0.020
AUFLIA/misc	14	10	0.010	14	0.022	**14**	0.021	14	0.023
AUFLIA/piVC	29	25	0.109	25	0.109	29	0.119	**29**	0.117
AUFLIA/RicAgla	14	14	0.052	**14**	0.050	14	0.050	14	0.050
AUFLIA/simplify	769	471	1.751	749	3.846	**762**	0.664	759	0.941
AUFLIRA/nasa	4619	**4113**	1.533	4113	1.533	4113	1.551	4113	1.533
AUFNIRA/nasa	142	46	0.044	**46**	0.043	46	0.043	46	0.044
Total	5599	4691	1.521	4973	1.849	**4990**	1.402	4987	1.409
Eager strategy		(i) BTBM		(ii) BTSM		(iii) STSM		(iv) IL	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	**12**	0.012	12	0.020	12	0.019	12	0.019
AUFLIA/misc	14	10	0.008	12	0.013	12	0.013	**14**	0.047
AUFLIA/piVC	29	25	0.107	25	0.108	29	0.127	**29**	0.106
AUFLIA/RicAgla	14	14	0.056	14	0.058	14	0.056	**14**	0.041
AUFLIA/simplify	769	25	18.24	24	39.52	497	30.98	**768**	0.739
AUFLIRA/nasa	4619	4527	0.072	**4527**	0.071	4527	0.074	4526	0.014
AUFNIRA/nasa	142	**72**	0.010	72	0.010	72	0.011	72	0.012
Total	5599	4685	0.168	4686	0.273	5163	3.047	**5435**	0.117

Table 1. Lazy vs. eager instantiation strategy in CVC3.

more sophisticated heuristics and so we have chosen to exclude them from the tables below. Also, there are 16 benchmarks that are known to be satisfiable, including all of the benchmarks in the AUFLIRA/misc family, so we have excluded them as well (we did not exclude any of the *nasa* benchmarks since we do not know for sure which of them are actually satisfiable).

For the remaining 5,599 benchmarks that could not be solved using the naive strategy, we tried the following instantiation strategies: (i) basic trigger/matching algorithm (BTBM) with none of the heuristics described in Section 5 (i.e. no multi-triggers, syntactic matching only); (ii) same basic triggers with the smarter matching (BTSM) described in Section 5.2; (iii) same as (ii) except with smart triggers (STSM) including multi-triggers as described in Section 5.1; and finally (iv) same as (iii) but with the instantiation level (IL) heuristic activated. The results are shown in Table 1. Each table lists the number of cases by family. Then, for each of the four strategies, and for each family, we list the number of cases successfully proved unsatisfiable and the *average* time spent on these successful cases.

As can be seen, the basic matching strategy is quite effective on about 4/5 of the benchmarks, but there are still nearly 1,000 that cannot be solved without more sophisticated techniques. Another observation is that the eager strategy generally outperforms the lazy strategy, both on average time taken and on number of cases proved. The notable exception is the *simplify* family. On this family, the lazy strategy performs much better for all except the very last column. This can be explained by the fact that the *simplify* benchmarks are especially sus-

		Vampire		SPASS		Simplify		CVC3	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
$T_\emptyset$	365	266	9.2768	302	1.7645	207	0.0679	**343**	0.0174
$T_{\forall,\rightarrow}$	6198	6080	2.1535	6063	0.6732	5957	0.0172	**6174**	0.0042
T_{prop}	1468	1349	4.3218	1343	1.0656	1370	0.0339	**1444**	0.0058
T_{eval}	1076	959	5.6028	948	0.7601	979	0.0423	**1052**	0.0077
T_{array}	2026	2005	1.4438	2000	0.2702	1943	0.0105	**2005**	0.0048
T_{array*}	14931	14903	0.6946	14892	0.2323	14699	0.0101	**14905**	0.0035
T_{policy}	1987	1979	1.4943	1974	0.2716	1917	0.0101	**1979**	0.0050
Total	28051	27541	1.5601	27522	0.4107	27072	0.0145	**27902**	0.0043

Table 2. ATP vs SMT

ceptible to getting lost due to looping. However, the lazy strategy is not subject to looping, so it does much better. This also explains why the last column is no better than the third column for the lazy strategy—in fact it's a bit worse, which we suspect is simply due to random differences in the order of instantiations. For the other benchmarks, however, eager instantiation is usually helpful and sometimes critical for finding the proof. This is especially true of the *nasa* families. Thus, the instantiation level heuristic can be seen as a way of combining the advantages of both the eager and lazy strategies. There is one *nasa* case which is particularly difficult and falls just inside the time limit for the first three columns and just outside the time limit in the last column. This is why one fewer *nasa* case is proved in the last column.

6.3 Comparison with ATP systems

One of our primary goals in this paper was to evaluate whether SMT solvers might be able to do better than ATP systems on real verification applications that require both quantifier and theory reasoning. The *nasa* benchmarks provide a means of testing this hypothesis as they are available in both TPTP and SMT-LIB formats (this was, in fact, one of the primary motivations for translating the benchmarks). We also translated the benchmarks into Simplify's format so as to be able to compare Simplify as well.

Table 2 compares CVC3 with Vampire, SPASS, and Simplify on these *nasa* benchmarks. For these tests, the timeout was 1 minute. We chose Vampire and SPASS because Vampire and SPASS are among the best ATP systems: Vampire is a regular winner of the CASC competitions [22], and SPASS was the best prover of those tried in [10]. For easier comparison to [10], the benchmarks are divided as in that paper into seven categories: $T_\emptyset$, $T_{\forall,\rightarrow}$, T_{prop}, T_{eval}, T_{array}, T_{policy}, T_{array*}. The first category $T_\emptyset$ contains the most difficult verification conditions. The other categories were obtained by applying various simplifications to $T_\emptyset$. For a detailed description of the categories and how they were generated, we refer the reader to [10]. We also exclude in this breakdown (as was also done in [10]) the 14 known satisfiable cases, so there are 28051 benchmarks in total.

The first observation is that all solvers can prove most of the benchmarks, as most of them are easy. The ATP systems do quite well compared to Simplify: while Simplify is generally much faster, both Vampire and SPASS prove more cases than Simplify. Since at the time these benchmarks were produced, Simplify was the only SMT solver that could support quantifiers, this can be seen as a validation of the choice of ATP systems over SMT solvers at the time.

However, CVC3 dominates the other systems in both time and number of cases solved. There are only 149 cases that CVC3 cannot solve (as mentioned above we suspect most of these are actually satisfiable) and the average time is less than a hundredth of a second. For the most challenging cases, those in $T_\emptyset$, CVC3 was able to solve 343 out of 365 cases, significantly more than the provers evaluated in [10] (the best system solved 280). At the time these tests were done, this was the best result ever achieved on these benchmarks. This supports our hypothesis that with the additional quantifier techniques introduced in this paper, modern SMT solvers may be a better fit for verification tasks that mix theory reasoning and quantifier reasoning.[15]

6.4 Comparison with other SMT systems

At the time of our experiments, we knew of only two other SMT systems that included support for both quantifiers and the SMT-LIB format: yices and Fx7. Yices was the winner of SMT-COMP 2006, dominating every category. Fx7 was a new system recently developed by Michal Moskal. Fx7 uses quantifier instantiation techniques that closely follow those used by Simplify, with some extensions [16]. Unfortunately, the quantifier reasoning techniques used in yices are not published, but our understanding is that it also uses extensions of the matching algorithms found in Simplify.

Table 3 compares Fx7, yices, and CVC3 on the same subset of benchmarks used in the first set of experiments. While yices is sometimes faster than CVC3, CVC3 can prove as many or more cases in every category. In total, CVC3 can prove 34 more cases than yices (yices does not support the AUFNIRA division, so we don't count the additional 72 cases CVC3 can prove in this division). Also, CVC3 is significantly faster on the *simplify* and *nasa* benchmarks.

We were also naturally very curious to know how CVC3 compares to Simplify. Results on the *nasa* benchmarks were given above. The other obvious set of benchmarks to compare on is the *simplify* benchmarks. Not surprisingly, Simplify can solve all of these benchmarks very fast: it can solve all 2,251 benchmarks in its suite in 469.05 seconds, faster than both yices and CVC3 which take much longer to solve just the 833 benchmarks that were translated into SMT-LIB format. However, Simplify achieves these impressive results by relying on special formula annotations that tell it which triggers to use. If these annotations are removed, Simplify can only prove 444 of the original 2251 benchmarks. Of

[15] It is worth mentioning that the majority of TPTP benchmarks do *not* contain significant theory reasoning and on these, ATP systems are still much stronger than SMT systems.

		Fx7		yices		CVC3	
Category	#cases	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	12	0.4292	**12**	0.0108	12	0.0192
AUFLIA/misc	14	12	0.6817	**14**	0.0500	14	0.0479
AUFLIA/piVC	29	15	0.5167	**29**	0.0300	29	0.1055
AUFLIA/RicAgla	14	14	0.6400	**14**	0.0257	14	0.0407
AUFLIA/simplify	769	760	3.2184	740	1.4244	**768**	0.7386
AUFLIRA/nasa	4619	4187	0.4524	4520	0.0824	**4526**	0.0138
AUFNIRA/nasa	142	48	0.4102	N/A	N/A	**72**	0.0118
Total	5599	5048	0.8696	5329	0.2681	**5435**	0.1168

Table 3. Comparison of SMT systems

course, it is a bit unfair to compare Simplify on these benchmarks with the annotations removed when both Simplify and the benchmarks were designed with the assumption that annotations would be used. In fact, it is likely that CVC3 would not perform as well if the benchmarks had not been crafted to be solvable by matching-based instantiation algorithms. On the other hand, the results on the *nasa* benchmarks show that the heuristics used by CVC3 can effectively be used to solve verification conditions where no effort has been made to provide annotations. The ability to prove such benchmarks *automatically* and without annotations represents a significant step forward for SMT solvers.

Ideally, we would have run Simplify on all of the SMT-LIB benchmarks. Unfortunately, Simplify does not read the SMT-LIB format and the translation from SMT-LIB to Simplify's language is non-trivial as it involves moving from a sorted to an unsorted language (translating the *nasa* cases into Simplify's format was easier because both TPTP and Simplify formats are unsorted).

7 Conclusion

In this paper, we presented new formalisms and techniques for quantifier reasoning in the context of satisfiability modulo theories. Significantly, our results indicate that these techniques make SMT solvers a better choice than ATP systems on some classes of verification conditions that make use of both theory reasoning and quantifiers. Our techniques are also competitive with other state-of-the art SMT solvers. Indeed, there are several benchmarks from the SMT-LIB library that have been solved for the first time using these techniques.

In future work, we plan to explore extensions of these techniques that allow for more substantial completeness claims. In particular, we plan to explore more sophisticated kinds of theory matching and integration of complete techniques such as quantifier elimination for those theories for which it is applicable.

References

1. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532.

Elsevier Science, 2001.

2. C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in SAT Modulo Theories. Technical Report 06-05, Department of Computer Science, The University of Iowa, Aug. 2006.
3. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
4. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.
5. C. W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories.* PhD thesis, Stanford University, Jan. 2003.
6. C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Proc. FroCoS '02*, volume 2309 of *LNAI*, pages 132–146. Springer, Apr. 2002.
7. P. Baumgartner and C. Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172:591–632, 2008.
8. L. de Moura. Private communication, 2006.
9. L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.
10. E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In D. A. Basin and M. Rusinowitch, editors, *IJCAR*, volume 3097 of *LNCS*, pages 198–212. Springer, 2004.
11. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
12. C. Flanagan, R. Joshi, and J. B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Intelligent Enterprise Technologies Laboratory, 2004.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, and J. B. Saxe. Extended static checking for Java. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 234–245, June 2002.
14. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
15. J. Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial Instantiation Methods for Inference in First Order Logic. *Journal of Automated Reasoning*, 28(4):371–396, 2002.
16. M. Moskal, J. Lopuszanski, and J. Kiniry. E-matching for fun and profit. In S. Krstic and A. Oliveras, editors, *Proceedings of the* 5^{th} *International Workshop on Satisfiability Modulo Theories (SMT '07)*, pages 25–35, July 2007. Berlin, Germany.
17. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.
18. D. A. Plaisted and Y. Zhu. Ordered Semantic Hyper Linking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.
19. V. Prevosto and U. Waldmann. SPASS+T. In G. Sutcliffe, R. Schmidt, and S. Schulz, editors, *Proceedings of ESCoR: Empirically Successful Computerized Reasoning, Seattle, WA*, volume 192 of *CEUR Workshop Proceedings*, pages 18–33, 2006.

20. A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.
21. M. E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.
22. G. Sutcliffe. The IJCAR-2004 Automated Theorem Proving Competition. *AI Communications*, 18(1):33–40, 2005.
23. G. Sutcliffe and C. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
24. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *CADE*, volume 2392 of *LNCS*, pages 275–279. Springer, 2002.