Finding Conflicting Instances of Quantified Formulas in SMT

Andrew Reynolds
The University of Iowa

Cesare Tinelli
The University of Iowa

Leonardo de Moura
Microsoft Research

***Abstract*—In the past decade, Satisfiability Modulo Theories (SMT) solvers have been used successfully in a variety of applications including verification, automated theorem proving, and synthesis. While such solvers are highly adept at handling ground constraints in several decidable background theories, they primarily rely on heuristic quantifier instantiation methods such as E-matching to process quantified formulas. The success of these methods is often hindered by an overproduction of instantiations which makes ground level reasoning difficult. We introduce a new technique that alleviates this shortcoming by first discovering instantiations that are in conflict with the current state of the solver. The solver only resorts to traditional heuristic methods when such instantiations cannot be found, thus decreasing its dependence upon E-matching. Our experimental results show that our technique significantly reduces the number of instantiations required by an SMT solver to answer "unsatisfiable" for several benchmark libraries, and consequently leads to improvements over state-of-the-art implementations.**

I. Introduction

Many recent formal methods applications rely heavily on Satisfiability Modulo Theories (SMT) solvers for answering logical queries required to solve complex tasks. These systems typically are composed of multiple cooperating decision procedures, or *theory solvers*, each specialized on sets of ground constraints over some background theory. Thanks to the widespread success and applicability of SMT solvers, there has been a push to use them to handle queries based on richer encodings that include quantified formulas. Handling such formulas in a general way has been an ongoing challenge in the SMT community.

To date, E-matching, first described in [13], is the most popular and successful method used by SMT solvers for handling quantified formulas. In this method, instances of a quantified formula are generated by matching selected terms in the formula (called *matching patterns*) with ground terms in the rest of the problem. While solvers based on E-matching have had widespread success over many applications, their power is often difficult to wield. One reason is that E-matching often produces a very large number of instances, which may exhaust a solver's memory or generally cause its performance to degrade. The problem is often compounded by instances that introduce new ground terms, which subsequently trigger even more instantiations. This can lead to non-terminating *matching loops* in the worst case, in which a repeating pattern of terms causes an infinite chain of instantiation steps.

It is thus important to limit the number of instances produced as a result of E-matching. Past research has addressed this issue in various ways, including the use of user-provided matching patterns (or *triggers*) [7], and methods for recognizing or avoiding matching loops [9]. We present a new quantifier instantiation procedure that aims at decreasing the number of produced instances by decreasing the dependency of SMT solvers on E-matching. This is done by looking for instantiations that lead directly to ground conflicts or to relevant new constraints. In this scheme, the solver resorts to E-matching only when it cannot perform instantiations of this sort. Our goal is to enable the sub-module that handles quantified formulas in a SMT solver to behave more like an efficient theory solver for ground constraints. In particular, our method enables the quantifier module to influence the search performed by the main engine by reporting conflicts and propagating relevant ground constraints, as typically done by efficient theory solvers based on the DPLL(T) [14] framework.

The instantiation procedure described in this paper applies to arbitrary SMT inputs containing quantified formulas. However, it is not intended to be a comprehensive solution for handling such formulas. Instead, it is meant to supplement existing instantiation techniques in a principled manner, so that those, such as E-matching, which are currently cumbersome and expensive, are invoked as little as possible.

1) Contributions: This paper presents a new technique for quantifier instantiation in DPLL(T)-based SMT solvers that on average significantly reduces the number of instantiations required to prove a formula unsatisfiable. We give a formal argument for various properties of the technique and the instances it produces. We describe an optimized implementation that is efficient in practice. Finally, we provide detailed evidence that our implementation leads to significant improvements, according to several metrics, over state-of-the-art SMT solvers handling quantified formulas.

2) Related Work: Various works have focused on methods for discovering the unsatisfiability of quantified formulas in SMT. The first implementation of E-matching was given in the solver Simplify [7], which included various techniques such as mod-time and pattern-element optimization. These techniques were used by the SMT solver Z3 [6] and enhanced further, as described in [5]. Quantifier instantiation in DPLL(T) as implemented in the SMT solver CVC3 [3] is described in [9]. Specifying decision procedures with quantified formulas through the use of triggers is described in [8]. Techniques also exist for discovering the satisfiability of quantified formulas in SMT, including reasoning in local theory extensions [11], complete instantiation [10] and finite model finding [15].

II. Formal Preliminaries

We assume the usual notions from many-sorted first-order logic with equality (denoted by $\approx$). We fix a set $\mathbf{S}$ of *sort*

ISBN: 978-0-9835678-4-4.

symbols and for every $S \in \mathbf{S}$ an infinite set of $\mathbf{X}_S$ of *variables of sort* S. We assume the sets $\mathbf{X}_S$ are pairwise disjoint and let $\mathbf{X}$ be their union. A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq \mathbf{S}$ of sort symbols and a set Σ^{f} of *(sorted) function symbols* $f^{S_1 \cdots S_n S}$, where $n \geq 0$ and $S_1, \ldots, S_n, S \in \Sigma^{\mathrm{s}}$. We drop the sort superscript from function symbols when it is clear from context or unimportant. We assume that signatures always include a Boolean sort Bool and constants $\top$ and $\bot$ of that sort (respectively, for true and false).

Given a many-sorted signature Σ, well-sorted terms, atoms, literals, clauses, and formulas with variables in $\mathbf{X}$ are defined as usual and referred to respectively as Σ-*terms*, Σ-*atoms* and so on.[1] A *ground term/formula* is a Σ-term/formula with no variables. When $\mathbf{x} = (x_1, \ldots, x_n)$ is a tuple of variables and Q is either $\forall$ or $\exists$, we write $Q\mathbf{x}\,\varphi$ as an abbreviation of $Qx_1 \cdots Qx_n\,\varphi$. If e is a Σ-term or formula and $\mathbf{x}$ has no repeated variables, we write $e[\mathbf{x}]$ to denote that e's free variables are from $\mathbf{x}$; if $\mathbf{s} = (s_1, \ldots, s_n)$ and $\mathbf{t} = (t_1, \ldots, t_n)$ are term tuples, we write $e[\mathbf{t}]$ for the term or formula obtained from e by simultaneously replacing each occurrence of x_i in e by t_i; we write $\mathbf{s} \approx \mathbf{t}$ for the set $\{s_1 \approx t_1, \ldots, s_n \approx t_n\}$.

A Σ-*interpretation* $\mathcal{I}$ maps: each $S \in \Sigma^{\mathrm{s}}$ to a non-empty set S^I, the *domain* of S in $\mathcal{I}$, with $\mathsf{Bool}^I = \{\top, \bot\}$; each $x \in \mathbf{X}$ of sort S to an element $x^{\mathcal{I}} \in I_S$; and each $f^{S_1 \cdots S_n S} \in \Sigma^{\mathrm{f}}$ to a total function $f^{\mathcal{I}} : S_1^I \times \cdots \times S_n^I \to S^I$. A satisfiability relation between Σ-interpretations and Σ-formulas is defined inductively as usual.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ is a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only for how it interprets the variables is also in $\mathbf{I}$) and isomorphism. A Σ-formula $\varphi[\mathbf{x}]$ is T-*satisfiable* (resp., T-*unsatisfiable*) if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas T-*entails* a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is T-*satisfiable* if $\Gamma \not\models_T \bot$. For a given signature Σ the *theory of equality (with uninterpreted functions)* or E, consists of the set of all Σ-interpretations. Informally, we refer to the sort and function symbols in this theory as *uninterpreted*.

A substitution σ is a mapping from variables to terms of the same sort, such that the set $\{x \mid \sigma(x) \neq x\}$, the *domain* of σ, is finite. We say that σ is a *grounding substitution* for a tuple $\mathbf{x} = (x_1, \ldots, x_n)$ of variables if σ maps each element of $\mathbf{x}$ to a ground term. If $\mathbf{t} = (t_1, \ldots, t_n)$, we write $\mathbf{x} \mapsto \mathbf{t}$ to denote the substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$; for a term or formula $e[\mathbf{x}]$, we write $e\sigma$ to denote the expression $e[\mathbf{t}]$. This notation extends to sets of formulas/terms as expected.

III. Finding Conflicts for Quantified Formulas

To handle quantified formulas, DPLL(T) solvers typically divide the input set of formulas into a set Q of quantified formulas and a set G of ground ones. To determine if $Q \cup G$ is unsatisfiable in the background theory T, they heuristically add to G selected ground instances of formulas from Q, and succeed when they have added enough instances to make G T-unsatisfiable. When G is T-satisfiable, they build a truth assignment for the atoms in G that satisfies all the formulas in G and is consistent with T. The truth assignment is represented as a set M of all the ground literals it satisfies, which we will call a *context*. In this case, a possible quantifier instantiation heuristic is to add, when possible, ground instances φ of formulas from Q that are in conflict with the current context M, in the sense that $M \cup \{\varphi\}$ is T-unsatisfiable. Adding such an instance to G will effectively force the solver to discard M and look for another context, if one exists.

This section presents a new quantifier instantiation procedure that, as described above, searches for instances of universally quantified formulas that are in conflict with the context maintained by the solver. For simplicity, we describe only a basic version of the procedure here. A more practical implementation is discussed in the next section.

For the rest of the section we fix a theory T of signature Σ, a Σ-formula $\forall\mathbf{x}\,\psi \in Q$ with $\psi[\mathbf{x}]$ quantifier-free, and a context M consisting of a T-satisfiable set of ground Σ-literals. We will use $\mathbf{T}_M$ to denote the set of all terms occurring in M.

A. Conflict Finding Instantiation Procedure

Our instantiation procedure tries to construct grounding substitutions σ for $\mathbf{x}$ such that $M \models_T \neg\psi\sigma$. We refer to σ as a *conflicting substitution for* (M, ψ). Conflicting substitutions are of interest since they suffice to show that there is no model of T that satisfies both M and $\forall\mathbf{x}\,\psi$.

Example 1: If M is $\{f(a) \not\approx g(b), b \approx h(a)\}$, then $\{x \mapsto a\}$ is a conflicting substitution for $(M, f(x) \approx g(h(x)))$. □

To simplify its presentation, we assume our procedure is run on the flat form of quantified formulas $\forall\mathbf{x}\,\psi$, defined as follows.

Definition 1: A flat form *of a quantified formula* $\forall\mathbf{x}\,\psi$ *is an equivalent formula* $\forall\mathbf{x}, \mathbf{y}\,(\mu \Rightarrow \varphi)$ *where*

- μ *is a conjunction of equalities* $x_0 \approx f(x_1, \ldots, x_n)$*, which we will call the* matching constraints*, where* $n \geq 0$ *and* $x_0, \ldots, x_n$ *are variables from* $\mathbf{x}, \mathbf{y}$*;*
- φ *is a quantifier-free formula, which we will call the* flattened body*, whose non-ground atoms are all equalities between variables from* $\mathbf{x}, \mathbf{y}$*.*

A flat form of $\forall\mathbf{x}\,\psi$ can be computed by starting with $\mu = \top$ and $\varphi = \psi$ and repeatedly replacing selected terms t in $\mu \Rightarrow \varphi$ by a fresh variable x_t and adding the equation $x_t \approx t$ to μ until all non-ground terms have the form x or $f(x_1, \ldots, x_n)$.

Definition 2: Let $\mathbf{z}$ *be a tuple of variables. An* assignment over $\mathbf{z}$ *is a set of equations of the form* $z \approx t$ *with* z *in* $\mathbf{z}$ *and* $t \in \mathbf{T}_M$*. A* constrained assignment over $\mathbf{z}$ *is a set* $E \cup C$ *where* E *is an assignment over* $\mathbf{z}$ *and* C *is a set of equalities and disequalities over* $\mathbf{z}$*. A constrained assignment* A *is* M*-*feasible *if* $M \cup A$ *is* T*-satisfiable.*

Given the context M and a flat form $\forall\mathbf{x}, \mathbf{y}\,(\mu \Rightarrow \varphi)$ of $\forall\mathbf{x}\,\psi$, our instantiation procedure will attempt to construct a constrained assignment A over the variables $\mathbf{x}, \mathbf{y}$ that summarizes the conditions under which one can build a conflicting

[1] In this formalization all atoms have the form $s \approx t$ with s and t of the same sort. Having $\approx$ as the only predicate symbol causes no loss of generality as other predicate symbols can be modeled as function symbols with return sort Bool.

proc $\mathsf{falsify}(\varphi_0, b_0)$
  if φ_0 is ground
    if $M \models_T \varphi_0 \Leftrightarrow \bar{b}_0$ **then** $\{\emptyset\}$ **else** $\emptyset$
  else if φ_0 is $x_1 \approx x_2$
    if b_0 is $\top$ **then** $\{\{x_1 \not\approx x_2\}\}$ **else** $\{\{x_1 \approx x_2\}\}$
  else if φ_0 is $\neg\varphi_1$ **then**
    $\mathsf{falsify}(\varphi_1, \bar{b}_0)$
  else if φ_0 is $\varphi_1 \vee \varphi_2$
    if b_0 is $\top$ **then**
      $\{A_1 \cup A_2 \mid A_1 \in \mathsf{falsify}(\varphi_1, b_0), A_2 \in \mathsf{falsify}(\varphi_2, b_0)\}$
    else
      $\mathsf{falsify}(\varphi_1, b_0) \cup \mathsf{falsify}(\varphi_2, b_0)$

Fig. 1. The falsify procedure. It returns a set $\mathcal{A}$ of constrained assignments such that $M \cup A \models_T (\varphi_0 \Leftrightarrow \bar{b}_0)$ for each $A \in \mathcal{A}$, where $\bar{b}_0$ denotes the complement of b_0.

proc $\mathsf{match}(S_0)$
  if S_0 is $\{y \approx f(\mathbf{z})\} \cup S_1$ **then**
    $\{A \cup \{y \approx f(\mathbf{t})\} \cup \mathbf{z} \approx \mathbf{t} \mid A \in \mathsf{match}(S_1),\ f(\mathbf{t}) \in \mathbf{T}_M\}$
  else
    $\{\emptyset\}$

Fig. 2. The match procedure. It returns a set $\mathcal{A}$ of constrained assignments such that $M \cup A \models_T S_0$ for each $A \in \mathcal{A}$.

substitution for (M, ψ). When it succeeds in building A, the procedure is also able to return one such substitution.

1) Quantifier Instantiation Procedure: A basic, unoptimized version of the procedure consists of three steps. The first step returns constrained assignments A which by construction falsify the flattened body φ; more precisely, constrained assignments A such that $M \cup A \models_T \neg\varphi$. The second step returns constrained assignments A' which by construction entail the matching constraints μ, that is, $M \cup A' \models_T \mu$. The third step considers unions of the constrained assignments $A \cup A'$ constructed in steps one and two, and tries to extract from $A \cup A'$ a grounding substitution $\mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$ such that $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A \cup A'$. If such a substitution exists, the procedure returns $\mathbf{x} \mapsto \mathbf{s}$ as a conflicting substitution for (M, ψ); otherwise, it fails. We discuss these three steps in more detail in the following.

a) Step 1: Construct constrained assignments conflicting with the flattened body φ: This step is executed by the recursive subprocedure falsify shown in Figure 1 (where for brevity we assume that the only Boolean connectives in φ are $\neg$ and $\vee$), which takes as input a subformula φ_0 of the flattened body φ, and a Boolean constant $b_0 \in \{\top, \bot\}$ indicating the polarity of φ_0 in φ, and returns a set of constrained assignments computed according to that polarity.[2] Its initial inputs are $(\varphi, \top)$.

b) Step 2: Construct constrained assignments that entail the matching constraints μ: This step constructs a set $\mathcal{A}$ of constrained assignments each of which entails μ. It does so by using the subprocedure match shown in Figure 2, which is called on the set of all the constraints S_μ in μ. For each matching constraint $z \approx f(z_1, \ldots, z_n) \in S_\mu$, the subprocedure considers all terms of the form $f(t_1, \ldots, t_n) \in \mathbf{T}_M$, and adds to A the constraints $z \approx f(t_1, \ldots, t_n), z_1 \approx t_1, \ldots, z_n \approx t_n$.

[2]Formula φ_0 has positive polarity in φ (indicated by $\top$) if and only if it occurs below an even number of $\neg$ symbols.

c) Step 3: Extract a conflicting substitution from constrained assignment: This step tries to generate a conflicting substitution for (M, φ), if there exists one. To do so, it considers all M-feasible constrained assignments $A' = A'_f \cup A'_m$, where $A'_f \in \mathsf{falsify}(\varphi, \top)$ and $A'_m \in \mathsf{match}(S_\mu)$. It partitions A' into two sets B' and C' such that the equivalence closure of B' contains at most one ground term per equivalence class. Using B', the procedure constructs a grounding substitution $\sigma = (\mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t})$, which we call a *completion of* A', by computing the equivalence closure of B', and then mapping every variable in the same equivalence class to the ground term in that class if there is one, or to an arbitrary one from $\mathbf{T}_M$ otherwise. If it succeeds in constructing a completion σ such that $M \models C'\sigma$, the procedure ends, returning the substitution $\mathbf{x} \mapsto \mathbf{s}$. Otherwise, it tries to extract a conflicting substitution from a different constrained assignment in $\mathcal{A}'$.

Example 2: To see how substitutions like σ above are computed, suppose T is E, the theory of equality, $M = \{f(a) \not\approx f(b)\}$, $B' = \{x \approx y, z \approx a, z \approx w\}$, and $C' = \{x \not\approx w\}$. Note that $A' = B' \cup C'$ is an M-feasible constrained assignment. The set B' induces the equivalence relation $\{\{x, y\}, \{w, z, a\}\}$. Adding b to the equivalence class of x leads to the grounding substitution $\sigma = \{x \mapsto b, y \mapsto b, z \mapsto a, w \mapsto a\}$ which is such that $M \models_{\mathrm{E}} C'\sigma$. □

We remark that, in our experience, guessing ground terms to add to the equivalence classes in the equivalence closure of B' in the third step of the procedure is rarely needed. The reason is that B' typically contains a *grounding equation* $z \approx t$ (with $t \in \mathbf{T}_M$) for each variable z in it. When this is not the case, it is because either z does not occur as an argument of a function symbol in the flattened form $\forall \mathbf{x}, \mathbf{y}\, (\mu \Rightarrow \varphi)$, or it is not relevant to the falsification of that formula.

We illustrate our procedure as a whole with a simple example where T is again the theory E of equality.

Example 3: Say M is $\{f(a) \not\approx g(b), b \approx h(a)\}$ and consider the formula $\forall x\, \psi$ where ψ is $f(x) \approx g(h(x))$. A flattened form of $\forall x\, \psi$ is

$$\forall x, y_1, y_2, y_3\ \underbrace{(y_1 \approx f(x) \wedge y_2 \approx h(x) \wedge y_3 \approx g(y_2))}_{\mu} \Rightarrow \underbrace{y_1 \approx y_3}_{\varphi}$$

If we run our procedure on this formula, $\mathsf{falsify}(y_1 \approx y_3, \top)$ returns the set of constrained assignments $\{\{y_1 \not\approx y_3\}\}$. The procedure then invokes $\mathsf{match}(S_\mu)$ where S_μ is $\{y_1 \approx f(x), y_2 \approx h(x), y_3 \approx g(y_2)\}$. The recursive calls of match when processing each equality in S_μ are as follows:

equation	output
	$\{\emptyset\}$
$y_3 \approx g(y_2)$	$\{\{y_3 \approx g(b), y_2 \approx b\}\}$
$y_2 \approx h(x)$	$\{\{y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a\}\}$
$y_1 \approx f(x)$	$\{\{y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a, y_1 \approx f(a)\}\}$

Let A' be the union of the (single) constrained assignments produced by falsify and match. Notice that A' is M-feasible. Splitting A' into $B' = \{x \approx a, y_1 \approx f(a), y_2 \approx h(a), y_3 \approx g(b)\}$ and $C' = \{y_2 \approx b, y_1 \not\approx y_3\}$, say, the procedure can generate (in this case only) the substitution $\sigma = \{x \mapsto a, y_1 \mapsto f(a), y_2 \mapsto h(a), y_3 \mapsto g(b)\}$. Since $M \models_{\mathrm{E}} C'\sigma$, the procedure returns the substitution $\{x \mapsto a\}$. Note that $M \models_{\mathrm{E}} f(a) \not\approx g(h(a))$, that is, $M \models_{\mathrm{E}} \neg\psi[a]$, which shows that the returned substitution is indeed conflicting. □

One can show by structural induction that the subprocedures falsify and match have the following properties.

Lemma 1: For all $A \in \mathsf{falsify}(\varphi, \top)$ and $A' \in \mathsf{match}(S_\mu)$, $M, A \models_T \neg\varphi$, and $M, A' \models_T \mu$.

Lemma 2: Let $\forall \mathbf{x}, \mathbf{y}\,(\mu \Rightarrow \varphi)$ be the flat form of $\forall \mathbf{x}\,\psi[\mathbf{x}]$. Let $A'_f \in \mathsf{falsify}(\varphi, \top)$, $A'_m \in \mathsf{match}(S_\mu)$, $A'[\mathbf{x}, \mathbf{y}] = A'_f \cup A'_m$, and $\sigma = \mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$. If $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T C$, then $M, \psi[\mathbf{s}] \models_T \neg(A' \setminus C)[\mathbf{s}, \mathbf{t}]$.

Proof: Let σ, A'_f, A'_m and A' be as above. By Lemma 1, $M, A'_f \models_T \neg\varphi$ and $M, A'_m \models_T \mu$. Thus, we have that $M, A' \models_T \mu \wedge \neg\varphi$ or, equivalently, $M, A' \models_T \neg(\mu \Rightarrow \varphi)$. By our assumption, we have that $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T C$. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t}, (A' \setminus C) \models_T \neg(\mu \Rightarrow \varphi)$ which implies that $M, (\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}] \models_T \neg(A' \setminus C)[\mathbf{s}, \mathbf{t}]$. The claim then follows by the equivalence of $(\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}]$ and $\psi[\mathbf{s}]$. ■

This justifies the correctness result for our procedure.

Proposition 1: Every substitution returned by the instantiation procedure is conflicting for (M, ψ).

Proof: Let σ, A'_f, A'_m and A' be as in Lemma 2. Recall our instantiation procedure in Step 3 partitions A' into $B' \cup C'$. We have that $M \models_T B'\sigma$ due to our construction of σ. Furthermore, by assumption the procedure returns σ only such that $M \models_T C'\sigma$. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A'$, and by Lemma 2 with $C = A'$, we have that $M, \psi[\mathbf{s}] \models_T \neg(A' \setminus A')[\mathbf{s}, \mathbf{t}]$, thus, $M, \psi[\mathbf{s}] \models_T \bot$. ■

2) Constraint-Inducing Substitutions: Even when no conflicting substitutions exist for (M, ψ), it may be useful to find other substitutions that help the solver deduce useful information about the terms in M. This can be done by relaxing one of the requirements on the substitutions returned by our instantiation procedure. Let $\sigma = \mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$ and $A' = B' \cup C'$ be as in Step 3 of the procedure, except that $M \models_T D\sigma$ does not hold for a non-empty subset $D \subseteq C'$. Since the proof of Lemma 2 does not rely on that entailment, we still have $M \cup \psi[\mathbf{s}] \models_T \neg D[\mathbf{s}, \mathbf{t}]$, even though σ is no longer conflicting for (M, ψ). We refer to σ as a *constraint-inducing substitution* for (M, ψ). If D is a conjunction of disequalities, we refer to σ as an *equality-inducing substitution* for (M, ψ). Observe that since each predicate symbol in D is applied to variables, and $\mathbf{s}$ and $\mathbf{t}$ are tuples of terms from $\mathbf{T}_M$, the entailed formula $\neg D[\mathbf{s}, \mathbf{t}]$ is a disjunction of constraints over terms in $\mathbf{T}_M$. As a consequence, it may be beneficial to generate the instance $\psi[\mathbf{s}]$ anyway since it causes the solver to deduce constraints over terms from $\mathbf{T}_M$. This contrasts with instantiations produced by E-matching, which often introduce constraints over fresh terms.

Example 4: Consider the quantified formula $\forall x\,\psi[x]$ from Example 3, and say M is $\{f(a) \approx c, d \approx g(b), b \approx h(a)\}$. Our procedure produces the same constrained assignment A' as in that example. In this case too, A' is M-feasible. However, the completion $\sigma = \{x \mapsto a, y_1 \mapsto f(a), y_2 \mapsto h(a), y_3 \mapsto g(b)\}$, corresponding to the partition $B' \cup C'$ of A' with $C' = \{y_2 \approx b, y_1 \not\approx y_3\}$, is *not* such that $M \models_{\mathrm{E}} (y_1 \not\approx y_3)\sigma$. In fact, it is not difficult to see there are no conflicting substitutions for ψ. However, M together with the instance $\psi[a]$, i.e. $f(a) \approx g(h(a))$, allows the solver to deduce that the terms $f(a)$ and $g(b)$ from $\mathbf{T}_M$ are equal. □

3) An Instantiation Strategy: A strategy can be used that produces both conflicting and constraint-inducing substitutions for a given context M and set of quantified formulas Q. First, if a conflicting substitution can be found for one quantified formula in Q, add the corresponding instance to the set of ground clauses G. This will cause the solver to backtrack some decision in M. Otherwise, if no conflicting substitution can be found, add instances corresponding to *every* constraint-inducing substitution found for each quantified formulas in Q.

IV. Practical Implementation

For greater clarity, the description of the instantiation procedure given in Section III favors simplicity over efficiency. Our actual implementation relies on one major restriction and numerous enhancements, briefly discussed in the following.

A. Restriction to the Theory of Equality

In our current implementation, the instantiation procedure does not reason modulo the actual background theory T but only modulo the theory E of equality. Concretely, this means that all function symbols in M and $\forall \mathbf{x}\,\psi$ (including arithmetic symbols) are treated as uninterpreted. This is done both for uniformity and efficiency since checking T-entailment/satisfiability is generally expensive for theories other than E. Since every theory T is a refinement of E (in the sense that it allows less interpretations), this restriction is sound: any conflicting substitution with respect to E is also conflicting with respect to a stronger theory. The obvious downside of this naïve approach is that for stronger theories the procedure returns only a coarse under-approximation of the set of conflicting substitutions for (M, φ).

Example 5: Let $M = \{f(a) \approx b, (g(a) \geq b{+}1) \approx \top\}$ and let $\forall \mathbf{x}\,\psi$ be $\forall x\, f(x) \approx g(x)$ where f, g, a, b are uninterpreted symbols and $\geq, +, 1$ are from the theory A of integer arithmetic. In this case, the background theory T is the union of E and A. Consider the following flat form of $\forall x\, f(x) \approx g(x)$:

$$\forall x, y_1, y_2\,(y_1 \approx f(x) \wedge y_2 \approx g(x)) \Rightarrow y_1 \approx y_2\ .$$

By treating the arithmetic symbols as symbols of E, our procedure will not discover any conflicting substitutions in this example. To see this, note that equating y_1 to $f(a)$ and y_2 to $g(a)$ in match (the only possibility) would produce the M-feasible constrained assignment $\{y_1 \not\approx y_2, y_1 \approx f(a), y_2 \approx g(a), x \approx a\}$. The corresponding substitution $\sigma = \{y_1 \mapsto f(a), y_2 \mapsto g(a), x \mapsto a\}$ is not conflicting for (M, ψ) in E because $M \not\models_{\mathrm{E}} (f(x) \not\approx g(x))\sigma$, so our current implementation of the procedure will return no substitutions in this case. In contrast, $M \models_{\mathrm{E}\cup\mathrm{A}} (f(x) \not\approx g(x))\sigma$ when $\geq, +, 1$ are treated as symbols of A. Hence, if our procedure did so and were able to determine the latter entailment it would be able to return the substitution $\{x \mapsto a\}$. □

We point out that reasoning modulo the actual background theory instead of E is not enough in general to return all possible conflicting substitutions, since the match sub-procedure is in fact incomplete for general theories T. To see this, observe that in A, an assignment containing $y \approx x + y_1$, $y_1 \approx 2$ will match with the term $3{+}2$, but fail to match with the equivalent term $2 + 3$. That said, for our purposes, using incomplete yet efficient theory matching and entailment tests may lead to the

proc $\mathsf{falsify}_i(\varphi_0, b_0, A, S)$
  if φ_0 is ground
    if $M \models_T \varphi_0 \Leftrightarrow \bar{b}_0$ **then** $\{(A, S)\}$ **else** $\emptyset$
  else if φ_0 is $x_1 \approx x_2$
    if b_0 is $\top$ **then**
      $\mathsf{match}_i(S\mid_{\{x_1,x_2\}}, A \cup \{x_1 \not\approx x_2\}, S \setminus S\mid_{\{x_1,x_2\}})$
    else
      $\mathsf{match}_i(S\mid_{\{x_1,x_2\}}, A \cup \{x_1 \approx x_2\}, S \setminus S\mid_{\{x_1,x_2\}})$
  else if φ_0 is $\neg\varphi_1$ **then**
    $\mathsf{falsify}_i(\varphi_1, \bar{b}_0, A, S)$
  else if φ_0 is $\varphi_1 \vee \varphi_2$
    if b_0 is $\top$ **then**
      $\bigcup_{(A',S') \in \mathsf{falsify}_i(\varphi_1, b_0, A, S)} \mathsf{falsify}_i(\varphi_2, b_0, A', S')$
    else
      $\mathsf{falsify}_i(\varphi_1, b_0, A, S) \cup \mathsf{falsify}_i(\varphi_2, b_0, A, S)$
proc $\mathsf{match}_i(S_0, A, S)$
  if S_0 is $\{y \approx f(\mathbf{z})\} \cup S_1$ **then**
    $S_0' := S_1 \cup S\mid_{\mathbf{z}}$; $S' := S \setminus S\mid_{\mathbf{z}}$;
    $\bigcup_{f(\mathbf{t}) \in \mathbf{T}_M} \mathsf{match}_i(S_0', A \cup \{y \approx f(\mathbf{t})\} \cup \mathbf{z} \approx \mathbf{t}, S')$
  else
    $\{(A, S)\}$

Fig. 3. The $\mathsf{falsify}_i$ and match_i procedures. We have that $M, A \models_T \neg((S_0 \setminus S) \Rightarrow \varphi_0)$ for each $(A, S) \in \mathsf{falsify}_i(\varphi_0, \top, \emptyset, S_0)$. $S\mid_V$ denotes the set of matching constraints from S whose left hand side is in V.

best performance, where conflicting substitutions are found only when it is reasonably easy for the procedure to do so.

B. Enhancements to the Basic Procedure

The most important enhancement with respect to the basic procedure described in Section III is that its three main steps are interleaved, as demonstrated in Figure 3. With respect to the basic procedure, $\mathsf{falsify}_i$ and match_i take two additional arguments: a constrained assignment A and a set of matching constraints S. Intuitively, A is the current constrained assignment we are building, and S is the matching constraints that are left to process. When considering a quantified formula with flat form $\forall \mathbf{x}.\, \mu \Rightarrow \varphi$, we initially call $\mathsf{falsify}_i$ with arguments $(\varphi, \top, \emptyset, S_\mu)$, where S_μ is the set of matching constraints from μ. This builds a set of pairs $\mathcal{A}$, such that for each $(A, S) \in \mathcal{A}$, we have $M, A \models_T \neg((S_\mu \setminus S) \Rightarrow \varphi)$. It can be shown that when $S \neq \emptyset$, the matching constraints in S do not need to be entailed when constructing a completion for A.

This procedure has several important advantages over the basic one. First, constrained assignments are built incrementally, which (although not shown here) allows us to discard a constrained assignment A as soon as it becomes M-infeasible. Second, matching constraints are processed for a variable x as soon as any constraint involving x is added to A, as in the second branch of $\mathsf{falsify}_i$ and in match_i, allowing us to eagerly determine cases where the current constrained assignment will not lead to a conflicting substitution. Third, we compute the set $\mathcal{A} = \mathsf{falsify}_i(\varphi, \top, \emptyset, S_\mu)$ lazily, which allows us to check whether there exists a conflicting substitution for a returned constrained assignment before producing the entire set $\mathcal{A}$.

C. Implementation Details

The ground theory solver maintains an equivalence relation $\equiv_M$ over the terms in $\mathbf{T}_M$ induced by the constraints in M (whereby $s \equiv_M t$ only if $M \models_{\mathrm{E}} s \approx t$). For each $t \in \mathbf{T}_M$, let $[t]_M$ denote the equivalence class of t in $\equiv_M$ and let $[\mathbf{t}]_M$ denote $([t_1]_M, \ldots, [t_n]_M)$ if $\mathbf{t} = (t_1, \ldots, t_n)$.[3] For every function symbol f of arity n in the input formula, we build an index $\mathcal{I}_f$ containing entries of the form $[\mathbf{t}]_M \mapsto f(\mathbf{t})$, mapping an n-tuple $[\mathbf{t}]_M$ of equivalence classes to some term $f(\mathbf{t}) \in \mathbf{T}_M$. The index is functional, that is, if $f(\mathbf{s}), f(\mathbf{t}) \in \mathbf{T}_M$ with $\mathbf{s} \equiv_M \mathbf{t}$ at most one of $f(\mathbf{s})$ and $f(\mathbf{t})$ is in $\mathcal{I}_f$. This data structure is used by the falsify procedure when checking entailment of ground equalities thanks to the following invariant maintained within the solver:

$$M \models_{\mathrm{E}} f(\mathbf{t}) \approx g(\mathbf{s}) \quad \text{iff} \quad \begin{cases} [\mathbf{t}]_M \mapsto f(\mathbf{u}) \in \mathcal{I}_f, \\ [\mathbf{s}]_M \mapsto g(\mathbf{v}) \in \mathcal{I}_g, \text{ and} \\ [f(\mathbf{u})]_M = [g(\mathbf{v})]_M. \end{cases}$$

To process matching constraints we build an extended index $\mathcal{J}_f$ with entries of the form $([f(\mathbf{t})]_M, [\mathbf{t}]_M) \mapsto f(\mathbf{t})$ for terms $f(\mathbf{t}) \in \mathbf{T}_M$. When considering a matching constraint $x \approx f(x_1, \ldots, x_n)$, the match procedure enumerates, modulo $\equiv_M$, the terms in $\mathbf{T}_M$ with top symbol f by traversing the index $\mathcal{J}_f$ — and backtracking whenever it determines that the constrained assignment it is constructing is not M-feasible.

Constrained assignments are represented as a pair (U, C), where U is a partial map from variables $\mathbf{x}, \mathbf{y}$ to a term they are equated to (either a representative term from $\mathbf{T}_M$ or another variable), and C is a set of flat constraints over $\mathbf{x} \cup \mathbf{y}$. Finally, formulas $\forall \mathbf{x}\, \psi$ are not actually flattened. Instead of replacing a term t in ψ with a fresh variable y, we treat t itself as y when needed.

V. Results

We implemented our instantiation procedure with the restrictions and enhancements mentioned in Section IV within the SMT solver CVC4 [1] (version 1.3). In this section, we compare the performance of our implementation against various state-of-the-art SMT solvers.[4]

We considered three different configurations of CVC4 that vary on the instantiation strategy they use. All of them apply quantifier instantiation *lazily*, that is, after the solver produces a T-satisfiable context M that propositionally satisfies the set G of current ground formulas. Given a set of *active* quantified formulas Q, each configuration of CVC4 runs one or more of the following steps in succession until a ground instance is added to G.

1) Add the instance $\psi[\mathbf{t}]$ if there exists a conflicting substitution $\mathbf{x} \mapsto \mathbf{t}$ for (M, ψ) for some $\forall \mathbf{x}\, \psi \in Q$.
2) Add the instances $\psi[\mathbf{t}]$ for a subset of the equality-inducing substitutions $\mathbf{x} \mapsto \mathbf{t}$ for (M, ψ), for each $\forall \mathbf{x}\, \psi \in Q$.
3) Add all instances based on E-matching for (M, Q).

The first configuration, which we will refer to as **cvc4**, performs Step 3 only. The second configuration, **cvc4+c**, performs Step 1 and Step 3. The third, **cvc4+ci**, performs all three steps. In Step 2, configuration **cvc4+ci** considers at most one equality-inducing substitution for each constrained assignment

[3] In the implementation, $[t]_M$ is represented by a distinguished term in it.
[4] Details can be found at http://cvc4.cs.nyu.edu/papers/FMCAD2014-qcf/.

TABLE I. NUMBER OF SOLVED UNSATISFIABLE BENCHMARKS.

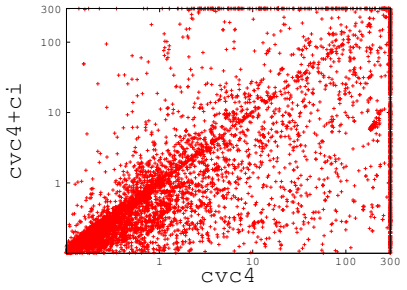
Set	Class	**cvc3**	**z3**	**cvc4**	**cvc4+c**	**cvc4+ci**
TPTP	**EPR**	596	**840**	809	768	769
	NEQ	910	**1,406**	1,346	1,374	1,373
	PEQ	641	656	668	690	**824**
	SEQ	3,087	3,366	3,277	3,581	**3,650**
	Sub-Total	5,234	6,268	6,100	6,413	**6,616**
Isabelle	**ArrowOrder**	321	178	307	339	**371**
	FFT	**296**	277	288	291	288
	FTA	**1,124**	917	990	1,012	1,018
	Hoare	607	549	563	579	**621**
	NS_Shared	105	108	117	140	**143**
	QEpres	297	325	360	361	**362**
	StrongNorm	207	241	242	251	**253**
	TwoSquares	643	620	708	712	**719**
	TypeSafe	227	291	283	298	**307**
	Sub-Total	3,827	3,506	3,858	3,983	**4,082**
SMT-LIB	**boogie**	653	**741**	678	692	706
	simplify	2,070	**2,478**	2,334	2,358	2,360
	why	380	**385**	369	371	373
	other	304	**379**	299	300	308
	Sub-Total	3407	**3,983**	3,680	3,721	3,747
	Total	12,468	13,757	13,638	14,117	**14,445**

produced by the first two steps of our instantiation procedure; that is, it does not add instances for multiple completions of the same constrained assignment. Configurations **cvc4+c** and **cvc4+ci** use the naïve approach for handling interpreted theory symbols described in Section IV-A. A single run of these steps we will refer to as an *instantiation round*.

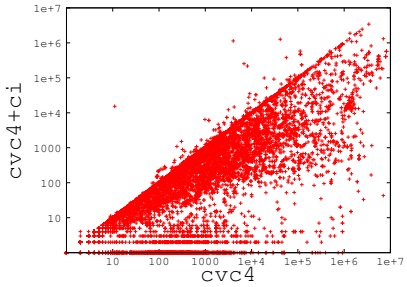
A. Comparison with SMT solvers

We compared these three configurations of CVC4 with the SMT solvers Z3 (version 4.3.2) [6] and CVC3 [3], both of which rely on quantifier instantiation to reason about quantified formulas. We report results on *unsatisfiable* benchmarks from various collections from the verification and automated theorem proving communities: the TPTP library (version 6.0.0) [17]; a set of benchmarks produced as proof obligations from Isabelle [4]; and SMT-LIB [2]. We considered 12,406 unsatisfiable benchmarks from TPTP which contain primarily quantified formulas and are all over the theory of equality.[5] We considered 13,041 Isabelle benchmarks (many of whom are classified as satisfiable or unknown) which also primarily contain quantified formulas, but also include both integer and real arithmetic constraints. Many of the SMT-LIB benchmarks represent software verification conditions, and make heavy use of symbols over several theories. We considered all 26,320 benchmarks from SMT-LIB that contained quantified formulas but no non-linear arithmetic constraints, which CVC4 does not yet support. Of all of these SMT-LIB benchmarks, we report results only for the 4,633 that were *non-trivial*, which we define here as taking more than 0.1 seconds to solve for at least one configuration of one solver. We ran all the experiments with a 300 second timeout per benchmark and analyzed the results according to two metrics: the performance of all solvers in terms of time and number of (unsatisfiable) benchmarks solved, and their efficiency in terms of the number of instantiations needed to answer unsatisfiable.

1) Problems Solved: Table I reports the number of benchmarks solved by the solvers for the three benchmark sets. For TPTP benchmarks, **cvc4+ci** is the overall winner, solving



(a) Runtime (in seconds).



(b) Reported number of instances.

Fig. 4. **cvc4+ci** vs **cvc4** over all benchmarks. Data shown on a log-log scale.

6,616 within the time limit. This is 347 more than **z3** and 516 more than **cvc4**. At least one configuration of CVC4 solves 34 unsatisfiable problems from TPTP with current rating 1.0, which is given to benchmarks that no ATP system can solve. In particular, 15 of these problems were solved using the new techniques (configurations **cvc4+c** and **cvc4+ci**) only. For Isabelle benchmarks, **cvc4+ci** is again the overall winner, solving noticeably more problems than the other solvers (4,082 vs. 3,858 for **cvc4**, 3,827 for **cvc3**, and 3,506 for **z3**). This shows that our techniques are quite effective on problems with mostly uninterpreted symbols. For SMT-LIB benchmarks, **z3** is the clear winner, with 3,983 solved problems. The new techniques yield a small improvement in performance, as **cvc4+ci** solves 67 more problems than **cvc4**. However, their performance still trails **z3**'s significantly, by 236 benchmarks. We conjecture that this is partially due to the fact that our procedure handles interpreted symbols naïvely, although several implementation differences exist between CVC4 and Z3.[6]

Overall, over the three benchmark sets, **cvc4+ci** solves more problems than any other configuration. In particular, it consistently outperforms **cvc4+c** (14,445 vs. 14,117), solving 404 problems that **cvc4+c** cannot, while **cvc4+c** only solves 76 that **cvc4+ci** cannot. This shows that computing constraint-inducing substitutions in addition to conflicting substitutions is beneficial. The scatter plot in Figure 4(a) shows that the new instantiation techniques (**cvc4+ci**) typically improve the runtime performance of CVC4—although there are several cases where they do not. Over the benchmarks they both solve, **cvc4+ci** solves 4,419 benchmarks at least 20% faster than **cvc4**, whereas **cvc4** solves 1,845 benchmarks at least 20% percent faster than **cvc4+ci**. We believe the improvement in performance is due to the reduction in the number of instances produced by **cvc4+ci**, as discussed later. Over all

[5]We did not consider TPTP benchmarks having TFF syntax (which includes theory constraints), since Z3 and CVC3 do not have a parser for this format, and no translator from this format was available.

[6]In particular, CVC4 does not use eager quantifier instantiation, clause deletion, or relevancy (see Section 7 of [5]) for SMT-LIB benchmarks.

ISBN: 978-0-9835678-4-4.

TABLE II. NUMBER OF REPORTED INSTANTIATIONS FOR SOLVED UNSATISFIABLE BENCHMARKS.

	TPTP		Isabelle		SMT-LIB	
	Solved	Inst	Solved	Inst	Solved	Inst
cvc3	5,245	627.0M	3,827	186.9M	3,407	42.3M
z3	6,269	613.5M	3,506	67.0M	**3,983**	6.4M
cvc4	6,100	879.0M	3,858	119.0M	3,680	60.7M
cvc4+c	6,413	190.8M	3,983	54.0M	3,721	41.0M
cvc4+ci	**6,616**	150.9M	**4,082**	28.2M	3,747	32.4M

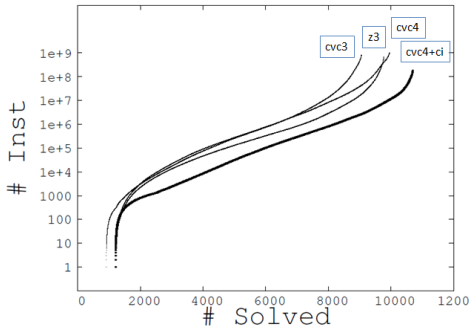
benchmark sets, **cvc4** solves 235 that **cvc4+ci** cannot solve, while **cvc4+ci** solves 1,042 benchmarks that **cvc4** cannot. At least one configuration of either **cvc4+ci** or **cvc4+c** solves 359 benchmarks that no implementation of E-matching (either Z3, CVC3, CVC4) can solve, indicating that our techniques can be used to improve the precision of SMT solvers for unsatisfiable problems containing quantified formulas.

2) Instances Generated: Table II gives the cumulative number of generated instances reported by each solver for the three benchmarks sets. For both the TPTP and the Isabelle set, in addition to solving the most benchmarks, configuration **cvc4+ci** requires by far the least number of instantiations to do so. For TPTP, **cvc4+ci** produces about 151 million instances to solve 6,616 problems, which is 5.8 times fewer than what **cvc4** requires for solving 6,100 problems. Similarly for Isabelle, **cvc4+ci** requires 28M instantiations to solve 4,082 problems, which is 4.2 times fewer than what **cvc4** requires for solving 3,858 problems. For SMT-LIB, **z3** is by far the most efficient solver, solving 3,983 problems while requiring only 6.4M instantiations. The new techniques in CVC4 reduce the instantiations by approximately half, which is less dramatic than the improvements seen on TPTP and Isabelle. This is again likely due to the prevalence of theory symbols in the encodings used by SMT-LIB benchmarks.

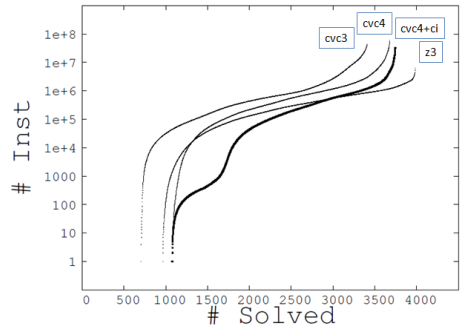
The scatter plot in Figure 4(b) compares the reported number of instances produced by configurations **cvc4** and **cvc4+ci** on the benchmarks they both solve. The plot clearly shows that **cvc4+ci** consistently requires many fewer instantiations, confirming that the instances it produces are generally effective at contributing towards finding refutations.

Figure 5 shows the cumulative number of instances reported by each of the solvers on the benchmarks they solve. For benchmarks with low theory content (from the TPTP and Isabelle libraries), the configuration **cvc4+ci** consistently produces fewer instances while solving more benchmarks than the other solvers and configurations. For SMT-LIB benchmarks, the plot shows that the configuration **cvc4+ci** uses considerably fewer instances than **z3** to solve its first 1,750 benchmarks. However, **cvc4+ci** requires more instantiations overall to solve fewer benchmarks than **z3**. This suggests that our techniques are highly effective at handling a subset of the SMT-LIB benchmarks, but require further enhancements to account for the encodings used by these benchmarks.

Table III shows a detailed view of the instances produced by the three configurations of CVC4. The first column (IR) gives the cumulative number of instantiation rounds each configuration requires for the benchmarks it solves. The remaining six columns give the percentage of instantiation rounds where they produce instances based respectively on E-matching, conflicting substitutions, and constraint-inducing substitutions; and the total number of instances produced for each of



(a) On TPTP and Isabelle benchmarks.



(b) On SMT-LIB benchmarks.

Fig. 5. Cactus plot showing the cumulative number of instantiations reported by all solvers on the benchmarks they solve.

TABLE III. DETAILS ON INSTANCES PRODUCED BY THREE CONFIGURATIONS OF CVC4.

		E-matching		Conflicting Sub.		C-Inducing Sub.	
	IR	%IR	# Inst	%IR	# Inst	%IR	# Inst
TPTP							
cvc4	71.6K	100.0	879.0M				
cvc4+c	202.0K	21.7	190.6M	78.3	158.2K		
cvc4+ci	209.0K	20.3	150.4M	76.4	159.7K	3.3	415.8K
Isabelle							
cvc4	7.0K	100.0	119.0M				
cvc4+c	18.2K	28.9	54.0M	71.1	12.9K		
cvc4+ci	21.8K	22.4	28.2M	64.0	13.9K	13.6	130.9K
SMT-LIB							
cvc4	14.0K	100.0	60.7M				
cvc4+c	51.7K	24.3	41.0M	75.7	39.1K		
cvc4+ci	58.0K	20.0	32.3M	71.6	41.5K	8.4	51.5K

these types. We can see that while configurations **cvc4+c** and **cvc4+ci** require significantly more instantiation rounds on average to answer unsatisfiable on each benchmark library, they require much fewer instances overall. Overall, a conflicting substitution was found on 77.3% of the instantiations rounds performed by **cvc4+c** and on 74.5% of the instantiation rounds performed by **cvc4+ci**. These percentages are fairly consistent across the three benchmark classes, indicating that a majority of satisfying assignments found at the ground level can be ruled out by an instance from a conflicting substitution. For **cvc4+ci**, a conflicting substitution was found on 78.5% of the instantiation rounds where a constraint-inducing substitution was not produced, which is slightly higher than the percentage found by **cvc4+c** alone (77.3%). This suggests that constraint-inducing substitutions help the solver find conflicting substitutions. In total, E-matching was called 1.57 fewer times by **cvc4+ci** than by **cvc4**, which led to a factor of 5 fewer instances produced as a result of such calls.

Overall, 12,165 of the 14,445 benchmarks that **cvc4+ci** solved required at least one instantiation round by all configurations of CVC4, and 2,520 of these 12,216 benchmarks (20.7%) could be solved by **cvc4+ci** using *only* instances resulting from conflicting and constraint-inducing substitutions. In other words, for 20.7% of the benchmarks it solves, **cvc4+ci** did not rely on E-matching at all to answer unsatisfiable. Moreover, 94 of these 2,251 benchmarks could not be solved by **cvc4** within the timeout, showing that difficult benchmarks can be solved solely by the techniques mentioned in this paper.

B. Comparison with Automated Theorem Provers

We do not give a detailed comparison with automated theorem provers, which are capable of handling benchmarks from the TPTP library, but do so using entirely different methods than SMT solvers. For a brief and informal overview, a recent (multi-strategy) run script for iProver [12] solves 6,508 unsatisfiable benchmarks from the TPTP library, while a recent run script for E [16] solves 9,751. A version of both of these scripts as well as the systems themselves were used in CASC 24, the latest competition for automated theorem provers. Using a run script devised for a similar purpose, which incorporates several configurations of E-matching as well as the techniques described here, CVC4 solves 7,227 unsatisfiable TPTP benchmarks, making CVC4 highly competitive with a state-of-the-art instantiation-based prover like iProver.

VI. Conclusion

We have presented a technique for quantifier instantiation in SMT that increases the ability of an SMT solver to detect unsatisfiable problems containing quantified formulas. The method relies on a more principled heuristic for choosing instances, focusing on those that communicate conflicts or relevant constraints to the ground-level sub-solver. It handles any set of quantified formulas by treating theory symbols (at worst) as uninterpreted. Our experiments show that the number of instantiations necessary to solve unsatisfiable benchmarks is on average decreased by almost an order of magnitude when compared to implementations using E-matching only. As a result, our implementation shows a noticeable improvement in performance in terms of average runtime and overall number of unsatisfiable benchmarks solved.

In future work, we plan to implement a more incremental version of our instantiation procedure to recognize conflicts while the SMT is reasoning at the ground level, which has been shown to lead to performance improvements in other implementations of quantifier instantiation in SMT [5], [9]. We also plan to extend the procedure beyond its naïve treatment of interpreted symbols to increase the number of conflicting substitution found for formulas containing such symbols. As discussed in Section IV-A, doing so requires devising fast, if incomplete, T-satisfiability tests for theories other than equality. Finally, we would like to identify language fragments and investigate extensions of our techniques that are *complete*, that is, guaranteeing the existence of a model for the input set when they fail to produce additional instances. A main challenge for this will be to ensure that the extension is also as efficient (or better) than competitive implementations of E-matching when the input problem is unsatisfiable.

Acknowledgements

We would like to thank Tim King for implementing preliminary infrastructure in CVC4 for extending these techniques to quantified formulas containing linear arithmetic.

References

[1] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of CAV'11*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

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

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[4] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Børner and V. Sofronie-Stokkermans, editors, *Automated Deduction*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011.

[5] L. de Moura and N. Bjørner. Efficient E-Matching for SMT solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[6] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, J. ACM, 2003.

[8] C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Reasoning with triggers. In P. Fontaine and A. Goel, editors, *SMT 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2013.

[9] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE-21), Bremen, Germany*, volume 4603 of *Lecture Notes in Computer Science*, pages 167–182. Springer, 2007.

[10] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *Proceedings of CAV'09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[11] C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 265–281. Springer, 2008.

[12] K. Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer Berlin Heidelberg, 2008.

[13] C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.

[14] 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.

[15] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 640–655. Springer Berlin Heidelberg, 2013.

[16] S. Schulz. E-a brainiac theorem prover. *Ai Communications*, 15(2):111–126, 2002.

[17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.