Finding Conflicting Instances of Quantified Formulas in SMT

Andrew Reynolds[1], Cesare Tinelli[1], and Leonardo de Moura[2]

[1] Department of Computer Science, The University of Iowa
[2] Microsoft Research

Abstract. Modern SMT solvers primarily rely on heuristic methods such as E-matching for answering "unsatisfiable" in the presence of quantified formulas. The success of these solvers is often hindered by an overabundance of instantiations produced by these heuristics, making it difficult for the solver to continue its operation. In this paper, we introduce new techniques that alleviate this shortcoming by first discovering instantiations that are in conflict with the current state of the solver. In these techniques, the solver only resorts to heuristic methods when such an instantiation cannot be found, thus decreasing its dependence upon E-matching. Our experimental results show our technique significantly reduces the number of instantiations required by the SMT solver for answering "unsatisfiable" for several benchmark libraries, and consequently leads to improvements over state-of-the-art implementations.

1 Introduction

To date, E-matching, first described in [14], is the most popular and successful method used by DPLL(T)-based 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 in automated reasoning, the power of these solvers is often difficult to wield. One reason is that E-matching often produces a large number of instances, which may cause a solver's performance to degrade or consume all of its memory. The problem is often compounded by instantiations that introduce new ground terms which subsequently trigger more instantiations. This can lead to non-terminating *matching loops* in the worst case, where a repeating pattern of terms causes an infinite chain of instantiations.

It is thus important to limit the number of instances produced as a result of E-matching. Previous research has tried to address this issue in various ways, including the use of user-provided matching patterns (or *triggers*) [7, 13], 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 DPLL(T) 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 in DPLL(T) [15].

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 existing techniques, such as E-matching, which are currently cumbersome and expensive, are invoked only as little as possible.

Related Work Various works have focused on methods for answering "unsatisfiable" for quantified formulas in SMT. An 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 moreover were further optimized, as described in [5]. Quantifier instantiation in DPLL(T) as implemented in the SMT solver CVC3 [3] is described in [9]. The use of triggers for quantified formulas is described in [13]. Specifying decision procedures with quantified formulas, in particular through the use of triggers, is described in [8]. Techniques also exist for answering "satisfiable" for quantified formulas in SMT, including reasoning in local theory extensions [11], complete instantiation [10] and finite model finding [16].

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 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.[3] A *ground term/formula* is a Σ-term/formula with no variables. When $\mathbf{x} = (x_1, \ldots, x_n)$ is 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.

[3] 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 is with loss of generality as other predicate symbols can be represented as function symbols with return sort Bool.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ 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 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 will write $\mathbf{x} \mapsto \mathbf{t}$ to denote the substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. For any term or formula $e[\mathbf{x}]$, we will also write $e\sigma$ to denote the expression $e[\mathbf{t}]$. The notation above extends to sets of formulas/terms as expected.

2 Finding Conflicts for Quantified Formulas

Typical approaches for handling quantified formulas in DPLL(T) solvers divide the input set of formulas into a set Q of quantified formulas and a set G of ground ones. They try to prove that $Q \cup G$ is unsatisfiable in the background theory T by adding to G selected ground instances of formulas from Q and determining the T-satisfiability of G. To do the latter, the SMT solver tries to find incrementally a truth assignment to the atoms in G that satisfies all the formulas in G and is consistent with T. A truth assignment is represented by a set M of ground literals, which we will call a *context.* A possible quantifier instantiation heuristic is to generate instances φ of formulas in Q that are in conflict with M, in the sense that $M \cup \{\varphi\}$ is T-unsatisfiable. Adding those instances to G will effectively force the solver to discard M and look for another context. The final goal is to add enough instances in G to make it T-unsatisfiable.

In this section, we present a new quantifier instantiation procedure that looks for ground instances that are in conflict with the current context. 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 (and sub-terms) occurring in M.

2.1 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 in no model of T that satisfies both M and $\forall \mathbf{x}\, \psi$.

Example 1. Say M is $\{f(a) \not\approx g(b), b \approx h(a)\}$. The substitution $\{x \mapsto a\}$ is a conflicting substitution for $(M, \forall x\, f(x) \approx g(h(x)))$.

To simplify the presentation of our procedure, we assume that it is not run directly on formulas like $\forall \mathbf{x}\, \psi$ above, but on flat forms of them defined as follows.

Definition 1. A *flat form* of a quantified formula $\forall \mathbf{x}\, \psi$ is an equivalent formula of the form $\forall \mathbf{x}, \mathbf{y}\, (\mu \Rightarrow \varphi)$ 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 equalities between variables from $\mathbf{x}, \mathbf{y}$.

A flat form of $\forall \mathbf{x}\, \psi$ can be readily computed by starting with $\mu = \top$ and $\varphi = \psi$ and repeatedly replacing any non-ground term 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 $f(x_1, \ldots, x_n)$.

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 satisfiable in T.

Given the context M and a flat form $\forall \mathbf{x}, \mathbf{y}\, (\mu \Rightarrow \varphi)$ of $\forall \mathbf{x}\, \psi$, our quantifier 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 substitution for (M, ψ). When it succeeds in building A, the procedure is also able to return one such substitution.

Instantiation Procedure A basic, unoptimized version of the procedure consists of three main steps. The first step returns constrained assignments A which by construction falsify the flattened body φ; that is, it returns constrained assignments A such that $M \cup A \models_T \neg\varphi$. The second step extends them to constrained assignments A' which by construction also entail the matching constraints μ, that is, $M \cup A' \models_T \mu$. The third step tries to extract from one of these extended constrained assignments A' a grounding substitution $\mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$ such that $M,\, \mathbf{x} \approx \mathbf{s},\, \mathbf{y} \approx \mathbf{t} \models_T A'$. 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.

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. It returns a set of constrained assignments computed according to the polarity b_0 of the input formula φ_0. Its initial inputs are $(\varphi, \top)$.

```
proc falsify(φ0, b0)
  if φ0 is ground
    if M ⊨_T φ0 ⇔ b̄0 then {∅} else ∅
  else if φ0 is x1 ≈ x2
    if b0 is ⊤ then {{x1 ≉ x2}} else {{x1 ≈ x2}}
  else if φ0 is ¬φ1 then
    falsify(φ1, b̄0)
  else if φ0 is φ1 ∨ φ2
    if b0 is ⊤ then
      {A1 ∪ A2 | A1 ∈ falsify(φ1, b0), A2 ∈ falsify(φ2, b0)}
    else
      falsify(φ1, b0) ∪ falsify(φ2, b0)
```

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 $A \in \mathcal{A}$. The expression $\bar{b}_0$ denotes the complement of b_0.

```
proc match(S0, A0)
  if S0 is {y ≈ f(z)} ∪ S1 then
    {A ∪ {y ≈ f(t)} ∪ z ≈ t | A ∈ match(S1, A0), f(t) ∈ T_M}
  else
    {A0}
```

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 $A \in \mathcal{A}$.

Step 2: Extend to constrained assignments that entail matching constraints μ. This step considers only the M-feasible assignments $A \in \mathsf{falsify}(\varphi, \top)$ and extends each of them to a set $\mathcal{A}$ of constrained assignments each of which entails the matching constraints μ as well. It does so by using the subprocedure match shown in Figure 2, which is initially called on (S_μ, A) where S_μ is the set of all the constraints 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 $z \approx f(t_1, \ldots, t_n), z_1 \approx t_1, \ldots, z_n \approx t_n$.

Step 3: Extract a conflicting substitution from constrained assignment. This step tries to generate a conflicting substitution for (M, φ), if there exists one, from the set $\mathcal{A}'$ of M-feasible constrained assignments in $\bigcup_{A \in \mathsf{falsify}(\varphi, \top)} \mathsf{match}(S_\mu, A)$. To do so, it picks an $A' \in \mathcal{A}'$ and partitions it into two sets B' and C' such that the equivalence closure of B' contains at most one ground term. 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 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 in Step 3 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 $C' = \{x \not\approx w\}$ where x, y, z, w are variables. Note that $A' = B' \cup C'$ is an M-feasible constrained assignment for those variables. 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 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 in our experience because 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 of equality and uninterpreted functions E.

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 arrangements $\{\{y_1 \not\approx y_3\}\}$. The procedure then invokes $\mathsf{match}(S_\mu, \{y_1 \not\approx y_3\})$ 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**
	$\{\{y_1 \not\approx y_3\}\}$
$y_3 \approx g(y_2)$	$\{\{y_1 \not\approx y_3, y_3 \approx g(b), y_2 \approx b\}\}$
$y_2 \approx h(x)$	$\{\{y_1 \not\approx y_3, y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a\}\}$
$y_1 \approx f(x)$	$\{\{y_1 \not\approx y_3, y_3 \approx g(b), y_2 \approx b, y_2 \approx h(a), x \approx a, y_1 \approx f(a)\}\}$

The only constrained assignment A' returned by match 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 $\mathsf{falsify}$ and match have the following properties.

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

1. *$M, A \models_T \neg\varphi$, and*

2. $M, A' \models_T \mu$.

This justifies the following correctness result for our procedure.

Lemma 2. *Suppose the instantiation procedure is applied to the flat form* $\forall \mathbf{x}, \mathbf{y}\,(\mu \Rightarrow \varphi)$ *of* $\forall \mathbf{x}\, \psi[\mathbf{x}]$. *Let* $A'[\mathbf{x}, \mathbf{y}] = A'_1 \cup A'_2$ *be a constrained assignment produced by Step 2 of our procedure, and let* $\sigma = \mathbf{x} \mapsto \mathbf{s} \cup \mathbf{y} \mapsto \mathbf{t}$. *If* $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A'_1$, *then* $M, \psi[\mathbf{s}] \models_T \neg A'_2[\mathbf{s}, \mathbf{t}]$.

Proof: Let σ, A', A'_1 and A'_2 be as above and let A be the assignment returned by falsify that A' extends. By Lemma 1, $M, A \models_T \neg\varphi$ and $M, A' \models_T \mu$. Since $A \subseteq A'$, we have that $M, A' \models_T \mu \wedge \neg\varphi$ or, equivalently, $M, A'_1, A'_2 \models_T \neg(\mu \Rightarrow \varphi)$. By our assumption, we have that $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A'_1$. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t}, A'_2 \models_T \neg(\mu \Rightarrow \varphi)$ which implies that $M, (\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}] \models_T \neg A'_2[\mathbf{s}, \mathbf{t}]$. The claim then follows by the equivalence of $(\mu \Rightarrow \varphi)[\mathbf{s}, \mathbf{t}]$ and $\psi[\mathbf{s}]$. ■

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

Proof: Let σ and A' be as in Lemma 2. Recall that Step 3 of our procedure partitions A' into $B' \cup C'$. We have both that $M \models_T B'\sigma$ and $M \models_T C'\sigma$ by construction of σ. Hence, $M, \mathbf{x} \approx \mathbf{s}, \mathbf{y} \approx \mathbf{t} \models_T A'$, and by Lemma 2 with $A'_1 = A'$ and $A'_2 = \emptyset$, we have that $M, \psi[\mathbf{s}] \models_T \bot$. ■

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 our procedure, except that $M \models_T D\sigma$ does not hold for a non-empty subset $D \subseteq C'$. Since the proof of Proposition 1 does not rely on that entailment, we still have that $M \cup \psi[\mathbf{s}] \models_T \neg D[\mathbf{s}, \mathbf{t}]$, 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 and assert the instance $\psi[\mathbf{s}]$ as it nevertheless causes the SMT solver to deduce constraints over terms from $\mathbf{T}_M$. This contrasts to 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 $\psi[a]$, which is $f(a) \approx g(h(a))$, allows the solver to deduce that the terms $f(a)$ and $g(b)$ from $\mathbf{T}_M$ are equal. □

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 SMT solver to backtrack some decision in M. Otherwise, if no conflicting substitution can be found, add instances corresponding to *each* constraint-inducing substitution found for all quantified formulas in Q.

3 Practical Implementation

For greater clarity, the description of the instantiation procedure given in Section 2 favors simplicity over efficiency. Our actual implementation relies on one major restriction and numerous enhancements briefly discussed in the following.

3.1 Restriction to the Theory of Equality

In our current implementation, the instantiation procedure does not actually reason modulo the given 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 naive approach is that for theories other than E, 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, the current version of 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 the theory of equality because $M \not\models_{\mathrm{E}} (f(x) \not\approx g(x))\sigma$. So the current 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 that and was able to determine the latter entailment it would be able to return the substitution $\{x \mapsto a\}$. $\square$

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

3.2 Enhancements to the Basic Procedure

The first enhancement in our implementation with respect to the basic instantiation procedure described in Section 2 is that its three main steps are actually interleaved: once it computes an M-feasible constrained assignment A that falsifies $\neg\varphi$, the implemented procedure immediately tries to extend A to an assignment that satisfies μ and then extract a conflicting substitution from it, ending as soon as it succeeds. Furthermore, the procedure checks the M-feasibility of the current assignment as it extends it, discarding the extension if it makes the assignment M-infeasible and looking for another one.

Other enhancements are based on the fact that 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 $\mathbf{t} = (t_1, \ldots, t_n)$.[4] 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 falsify when checking entailment of ground equalities thanks to the following invariant maintained within the solver:

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

During match, we build an extended index $\mathcal{J}_f$ for terms $f(\mathbf{t}) \in \mathbf{T}_M$ with entries of the form $([f(\mathbf{t})]_M, [\mathbf{t}]_M) \mapsto f(\mathbf{t})$. When considering a matching constraint $x \approx f(x_1, \ldots, x_n)$, match 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.

[4] In the implementation $[t]_M$ is represented by a distinguished term in $[t]_M$.

4 Results

We implemented our instantiation procedure with the restrictions and enhancements mentioned in Section 3 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.

In our experiments, we used three different configurations of CVC4 that vary on the instantiation strategy they use. All three configurations 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 (a subset 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 the 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 produced by the procedure `match` from Section 2.1; that is, it does not add instances for multiple completions of the same constrained assignment. Configurations **cvc4+c** and **cvc4+ci** use the naive approach for handling interpreted theory symbols described in Section 3.1. A single run of these steps we will refer to as an *instantiation round*.

4.1 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 considered a set of *unsatisfiable* benchmarks from various benchmark collections from the verification and automated theorem proving communities: the TPTP library (version 6.0.0) [18], a set of benchmarks produced as proof obligations from Isabelle in [4], as well as SMT-LIB [2]. The TPTP benchmarks contain primarily quantified formulas and are all over the theory of equality. The Isabelle benchmarks 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 4,634 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

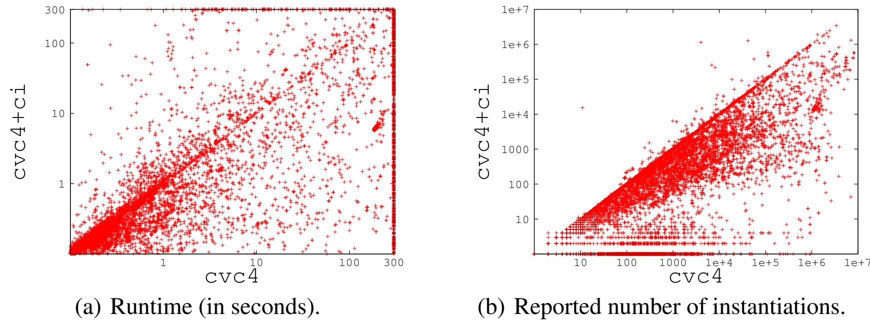
Set	Class	cvc3	z3	cvc4	cvc4+c	cvc4+ci
TPTP	**EPR**	596	**840**	809	768	769
	NEQ	910	**1,406**	1,346	1,374	1,373
	PEQ	641	656	668	690	**824**
	SEQ	3,087	3,366	3,277	3,581	**3,650**
	Total	5,234	6,268	6,100	6,413	**6,616**
Isabelle	**Arrow_Order**	321	178	307	339	**371**
	FFT	**296**	277	288	291	288
	FTA	**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**
	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
	Total	3407	**3,983**	3,680	3,721	3,747
	Cumulative Total	12,468	13,757	13,638	14,117	**14,445**

Fig. 3. Number of solved unsatisfiable benchmarks.

number of benchmarks solved, and their efficiency in terms of the number of instantiations needed to answer unsatisfiable.

Problems Solved Figure 3 reports the number of benchmarks solved by the SMT solvers for the three benchmark sets. For TPTP benchmarks, **cvc4+ci** is the overall winner, solving 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 quantifier-instantiation techniques is 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 lead to a small improvement in performance, as **cvc4+ci** solves 67 more problems than **cvc4**. However, the performance still trails **z3** significantly (by more than 200 benchmarks), which we partially attribute to the fact that our instantiation procedure handles interpreted symbols in a naive way.

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, showing 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 it does not. Over the benchmarks they both solve, **cvc4+ci** solves 4,419 benchmarks at least 20 percent faster than **cvc4**, whereas **cvc4** solves 1,845 benchmarks at least 20 percent faster than **cvc4+ci**. We believe the im-



(a) Runtime (in seconds). (b) Reported number of instantiations.

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

	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.M	3,680	60.7M
cvc4+c	6,413	190.8M	3,983	54.0M	3,721	41.1M
cvc4+ci	**6,616**	150.9M	**4,082**	28.2M	3,747	32.5M

Fig. 5. Number of reported instantiations for solved unsatisfiable benchmarks.

provement in performance is due to the reduction in the number of instantiations produced by **cvc4+ci**, as discussed later. Over all benchmark sets, **cvc4** solves 235 that **cvc4+ci** cannot, 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.

Instances Generated Figure 5 gives the cumulative number of instantiations 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 instantiations 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

			E-matching		Conflicting Sub.		C-Inducing Sub.	
		IR	% IR	# Inst	% IR	# Inst	% IR	# Inst
TPTP	**cvc4**	71,634	100.0	878,957,688				
	cvc4+c	201,990	21.7	190,607,350	78.3	158,162		
	cvc4+ci	208,970	20.3	150,351,384	76.4	159,696	3.3	415,772
Isabelle	**cvc4**	6,969	100.0	119,008,834				
	cvc4+c	18,160	28.9	53,969,845	71.1	12,918		
	cvc4+ci	21,756	22.4	28,196,846	64.0	13,932	13.6	130,864
SMT-LIB	**cvc4**	14,032	100.0	60,650,746				
	cvc4+c	51,696	24.3	40,954,033	75.7	39,145		
	cvc4+ci	58,003	20.0	32,305,788	71.6	41,531	8.4	51,454

Fig. 6. Details on instances produced by three configurations of CVC4. Column 1 (IR) gives the cumulative number of instantiation rounds each configuration requires for the benchmarks it solves. Column 2 gives the percentage of instantiation rounds where instances were added due to E-matching. Column 3 gives the number of instances produced by E-matching. Similarly, Columns 4 and 5 gives the percentage of rounds in which a conflicting substitution was produced and the number of instances added in this way. Columns 6 and 7 show the same for constraint-inducing substitutions.

shows that **cvc4+ci** consistently requires many fewer instantiations, confirming that the instantiations it produces are consistently effective at contributing towards finding refutations.

Figure 6 shows a detailed view of the instances produced by the three configurations of CVC4, giving the cumulative total of instantiation rounds performed by the configurations for the benchmarks they solve; 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 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 class, they require much fewer instantiations 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. This demonstrates that a majority of satisfying assignments found at the ground level can be ruled out by a single 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 towards finding 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 instantiations 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 for answering unsatisfiable. More-

over, 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.

4.2 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 [17] 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 provers like iProver.

5 Conclusion

We have presented a method for performing quantifier instantiation in SMT that increases the ability of an SMT solver to detect unsatisfiable problems with quantified formulas. The method relies on a more principled heuristic for choosing instances, in particular, by choosing those that communicate conflicts or relevant constraints to the ground-level sub-solver. It can be used to handle any set of quantified formulas by treating theory symbols (at worst) as uninterpreted. Our experiments show that the number of instantiations necessary for solving 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 an 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 improvements in performance in other implementations of quantifier instantiation in SMT [5, 9]. We also plan to extend the procedure beyond its naive treatment of interpreted symbols as uninterpreted to increase the number of conflicting substitution found for formulas containing such symbols. As discussed in Section 3.1, 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 is to ensure that the extension is also as efficient (or better) than competitive implementations of E-matching when the input problem is unsatisfiable.

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* 19th *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. M. Moskal, J. Łopuszański, and J. R. Kiniry. E-matching for fun and profit. *Electron. Notes Theor. Comput. Sci.*, 198(2):19–35, May 2008.
14. C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford, CA, USA, 1980. AAI8011683.
15. 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.
16. 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.
17. S. Schulz. E-a brainiac theorem prover. *Ai Communications*, 15(2):111–126, 2002.
18. 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.