EPiC Series in Computing

Volume 44, 2017, Pages 1–15

Vampire 2016. Proceedings of the 3rd Vampire Workshop

Conflicts, Models and Heuristics for Quantifier Instantiation in SMT

Andrew Reynolds[1]

Department of Computer Science, The University of Iowa, USA

Abstract

Satisfiability Modulo Theories (SMT) solvers have emerged as prominent tools in formal methods applications. While originally targeted towards quantifier-free inputs, SMT solvers are now often used for handling quantified formulas in automated theorem proving and software verification applications. The most common technique for handling quantified formulas in modern SMT solvers in quantifier instantiation. This paper gives an overview of recent advances in quantifier instantiation in SMT. In addition to the well-known technique known as E-matching, we discuss the use of conflicts and models for accelerating the search for (un)satisfiably. We further mention new instantiation-based techniques that are specialized to background theories such as linear real and integer arithmetic, and future work in this direction.

1 Introduction

The use of quantified formulas is highly important in a number of applications. In automated theorem proving applications [9, 10], quantified formulas often correspond to a set of background axioms, or to encode theories not natively supported by solvers. Quantified formulas also are used in software verification applications [31, 12], where they may be used for explicit function unfolding, or to reason about code by contracts. In synthesis, conjectures that specify universal properties for functions to synthesize are often expressed as formulas with one quantifier alternation. Finally, in planning applications, quantified formulas are sometimes used to specified that a desired plan must have certain properties for all relevant time points.

Unfortunately, the satisfiability problem for quantified formulas is highly challenging and only decidable in limited cases. For example, the satisfiability problem for the Bernays-Shonfinkel class of formulas having quantifier prenex $\exists^*\forall^*$ and no function symbols is NEXPTIME-complete; quantifier elimination algorithms can be used for deciding formulas e.g. in pure linear real arithmetic in time that is worst-case doubly exponential in the number of quantifier blocks. In spite of the theoretical challenges, many quantified formulas that arise in practice can be handled efficiently due to the development of efficient algorithms used by modern automated theorem provers.

A growing number of modern tools exist for reasoning about first-order quantified formulas. A number of automated theorem provers such as Vampire [30], E [44], and SPASS [48] were developed to target quantified formulas in single-sorted first-order logic. More recently, these tools have been extended with dedicated support for background theories [1, 7]. For the most part, tools from this community are superposition-based, although some are instantiation-based such as the Inst-Gen calculus used by iProver [22]. A recent approach known as AVATAR [37] leverages support in SAT and SMT solvers for quantifier-free formulas in combination with a superposition-based theorem prover.

In contrast to these tools, Satisfiability Modulo Theories (SMT) solvers such as CVC [46], Yices [19], and Z3 [15], were originally developed to target quantifier-free formulas in background theories, but have more recently been extended with support for universally quantified formulas. The approaches here are mostly instantiation-based based [17, 16, 23], although some approaches are based on superposition [14].

This paper overviews recent techniques for quantifier instantiation in SMT solvers. In Section 2, we outline an abstract procedure used by most SMT solvers that support quantified formulas. We then focus on three general-purpose instantiation strategies for handling quantified formulas in SMT:

- Section 3 will discuss *heuristic* techniques for quantifier instantiation based on pattern-matching that are widely-used by current solvers,
- Section 4 will discuss how finding *conflicts* can improve the ability of solvers to answer "unsat",
- Section 5 will examine how constructing candidate *models* can enable the solvers to answer "sat".

Section 6 introduces further techniques for quantifier instantiation in SMT that are specialized to theories. Finally, we conclude in Section 7 with a summary and mention possible future work.

2 Support for Quantified Formulas in an SMT Solver

Most modern SMT solvers are based on the DPLL(T) solving architecture, where a set of decision procedures for the background theory are modularly combined with a SAT solver for propositional satisfiability [36]. These solvers have also been extended in the past decade with approaches for universal and existential quantification [17, 16, 23]. For consistency, in this paper we assume all existential quantification is rewritten to universal quantification:

$$\exists \vec{x}\, P(\vec{x}) \rightsquigarrow \neg\forall \vec{x}\, \neg P(\vec{x})$$

This section gives an brief review of how DPLL(T)-based solvers check the satisfiability of inputs with quantified formulas, which can be summarized by the procedure $\mathsf{DPLL}_{\forall\mathsf{T}}$ in Figure 1. In this procedure, we first check whether this set is propositionally satisfiable using an underlying satisfiability (SAT) solver in Line 2. If Γ is propositionally unsatisfiable, then it is also unsatisfiable modulo T and we return "unsat" in line 3. Otherwise, in the case that Γ is satisfiable, the SAT solver will return a set of literals M that propositionally entails it. We partition M in line 4 into two parts E and Q, where E is quantifier-free and the atoms of literals in Q are universally quantified formulas. In line 5, we check the T-satisfiability of quantifier-free part E of M. If this is unsatisfiable, then we add the clause $\neg C$ (often referred to as a conflict or blocking clause) to Γ in line 6 and repeat, where C is an unsatisfiable subset of E. If the quantifier-free portion is satisfiable, we proceed to Line 7, which invokes the abstract procedure $\mathsf{check}_\forall$ on E and Q. This procedure either will determine that $E \cup Q$ is T-satisfiable and return the pair ("sat", $\emptyset$), or otherwise will return ("unknown", L), where L is a set of formulas that are valid in theory T (often referred to as T-lemmas), after which the procedure $\mathsf{DPLL}_{\forall\mathsf{T}}$ either terminates with "sat" or adds the set L to Γ and repeats.

Example 1. *Consider the set of* UFLIA*-formulas* Γ*:*

$$\{\neg P(a) \vee f(a) > a+1, P(a) \vee \forall x\, R(x) \vee (\forall y\, P(y) \vee f(y) < y)\}$$

On line 2, a SAT solver consider the propositional abstraction of this set:

$$\{\neg A_1 \vee A_2, A_1 \vee A_3 \vee A_4\}$$

```
DPLL$_{\forall T}(\Gamma)$:
    If $\Gamma$ is propositionally unsatisfiable,
        return "unsat".
    Otherwise, let $M = E \uplus Q$ be a set of literals such that $M \models_p \Gamma$.
    If $E$ is $T$-unsatisfiable,
        return DPLL$_{\forall T}(\Gamma \cup \neg C)$ for some $T$-unsatisfiable $C \subseteq E$.
    Otherwise, let $(r, L) =$ check$_\forall(E, Q)$.
    If $r$ is "unknown",
        return DPLL$_{\forall T}(\Gamma \cup L)$.
    Otherwise, return "sat".
check$_\forall(E, Q)$:
    Do one of the following:
        Return ("unknown", $L$) for some set $L$ of $T$-lemmas.
        Return ("sat", $\emptyset$), if $E \cup Q$ is $T$-satisfiable.
```

Figure 1: An abstract procedure for T-inputs Γ_0 with quantified formulas in an SMT solver. In Line 5, E is quantifier-free, and the atoms of each literal in Q are universally quantified formulas.

where for instance $\forall y\, P(y) \vee f(y) < y$ is abstracted as Boolean variable A_4. We have that e.g. $\{\neg A_1, A_3\}$ propositionally entails this set. Subsequently, line 4 considers the original atoms these variables correspond to, that is, $M = E \uplus Q = \{\neg P(a)\} \uplus \{\forall x\, R(x)\}$. On line 5, we check if E is satisfiable. Since E is T-satisfiable in this case, on line 7 we invoke check$_\forall$ *on (E, Q), which will either determine this set is satisfiable modulo* UFLIA*, or add additional formulas to* Γ. □

Support for quantified formulas in DPLL(T)-based SMT solvers depend primarily on how the function check$_\forall$ is implemented. We will examine such an implementation in this paper, in particular addressing the questions:

- What lemmas L should we return?
- How can we establish that $E \cup Q$ is T-satisfiable?

It is important to note that some SMT approaches to quantified formulas reason about quantified formulas *eagerly* during the DPLL(T) search [16]. In terms of Figure 1, these approaches invoke check$_\forall$ when M is incomplete and does not necessarily propositionally entail all formulas in Γ. The advantage of doing so is that lemmas returned by check$_\forall$ may help prune search space, while the disadvantage is that calling check$_\forall$ may be expensive and slow the search.

2.1 Skolemization and Instantiation

Common implementations of check$_\forall(E, Q)$ in SMT solvers return sets L consisting of T-lemmas of the following two forms:

1. $\neg\forall\vec{x}\, P(\vec{x}) \Rightarrow \neg P(\vec{k})$, where $\vec{k}$ is a set of fresh *Skolem* variables, and
2. $\forall\vec{x}\, P(\vec{x}) \Rightarrow P(\vec{t})$, where $\vec{t}$ is a tuple of ground terms.

Lemmas of the first type, which we refer to as *Skolemization lemmas*, witness the negation of universally quantified formulas for a fresh set of variables $\vec{k}$, are returned for each quantified formula whose negation occurs in Q. Notice that such lemmas need only be added once per quantified formula. Lemmas of the second type, which we refer to as *instantiation lemmas*, infer an instance of a universally quantified formula for some tuple of ground terms $\vec{t}$. In contrast, multiple lemmas of this form may be added per quantified formula. Unlike techniques for clause selection in automated theorem provers [45], typical strategies may add instantiation lemmas for *all* quantified formulas in Q simultaneously whenever check$_\forall$ is invoked. The performance of the solver is highly dependent on a having a good strategy for selecting such instantiation lemmas. We examine various strategies for selecting instantiation lemmas in Sections 3- 6.

3 E-matching

The most widely used technique for quantifier instantiation in SMT a heuristic technique known as *E-matching*. It was originally introduced in the Ph.D. thesis of Greg Nelson [35]. Recently, variants of the approach have been implemented in a number of solvers [17, 6, 15, 11, 4, 43]. The support for E-matching in these solvers is critical to the success of high-level tools for software verification [31, 12] and automated theorem proving [9, 10]. E-matching chooses instances for quantified formulas $\forall \vec{x}\, \varphi$ based on *pattern terms* whose free variables are $\vec{x}$. We will write $p[\vec{x}]$ to denote a term whose free variables are $\vec{x}$. In detail, E-matching does the following for each $\forall \vec{x}\, \varphi[\vec{x}]$ in Q:

1. Choose a set of patterns $p_1[\vec{x}], \dots, p_m[\vec{x}]$.
2. For each $j = 1, \dots, m$,
   (a) Compute a set of pairs $(\vec{t_1}, g_1), \dots, (\vec{t_n}, g_n)$ where for each $i = 1, \dots, n$, we have that g_i is a ground term from E such that $E \models_T g_i = p_j[\vec{t_i}]$.
   (b) Return the lemmas $\{\forall x\, \varphi[\vec{x}] \Rightarrow \varphi[\vec{t_1}], \dots, \forall x\, \varphi[\vec{x}] \Rightarrow \varphi[\vec{t_n}]\}$.

In other words, we first select a set of pattern terms, which are often some of the subterms of φ. For each of these patterns $p_j[\vec{x}]$, we compute a set of pairs of the form $(\vec{t_i}, g_i)$ where g_i is equivalent to $p_j[\vec{t_i}]$ under the assumption that E holds. We then return instantiation lemmas corresponding to each of these pairs. In the context of Figure 1, these lemmas are included in the return value of check$_\forall$ (set L). Notice in this description, we write $E \models_T g_i = p_j[\vec{t_i}]$ to denote that E entails $g_i = p_j[\vec{t_i}]$ modulo theory T. In typical E-matching implementations, T is limited to the theory of uninterpreted functions and equality.

Example 2. *Let E be the set $\{P(a), \neg P(b), R(c), \neg R(a), S(d)\}$ where a, b, c, d are free constants and P, R, S are unary predicates. Let ψ be $\forall x\, P(x) \vee R(x)$. Say we choose the pattern terms $P(x)$ and $R(x)$ for ψ. In Step 2, we compute pairs $(a, P(a))$ and $(b, P(b))$ for $P(x)$ and $(a, R(a))$ and $(c, R(c))$ for $R(x)$, and E-matching returns $\{\psi \Rightarrow P(a) \vee R(a), \psi \Rightarrow P(b) \vee R(b), \psi \Rightarrow P(c) \vee R(c)\}$.* $\square$

Example 3. *E-matching also takes into account equality and uninterpreted functions. For example, let E be the set $\{P(a,c), f(b) = a\}$, let ψ be $\forall xy\, P(f(x), y) \Rightarrow g(x) = y$, and let $P(f(x), y)$ be a pattern for it. We may consider the pair $((b,c), P(a,c))$, which is such that $E \models_{\text{EUF}} P(a,c) = P(f(b),c)$ since $f(b) = a \in E$, where $\models_{\text{EUF}}$ denotes entailment the theory of equality and uninterpreted functions. E-matching in this example returns $\{\psi \Rightarrow (P(f(b),c) \Rightarrow g(b) = c\}$.* $\square$

Recall in function check$_\forall$, we are interested in the (un)satisfiability of $E \cup Q$. As mentioned, E-matching selects instantiation lemmas based on pattern matching. It is typically effective for establishing unsatisfiability. The intuition for its effectiveness is the following. Let ψ be a formula in Q of the form $\forall \vec{x}\, \varphi[p[\vec{x}], \vec{x}]$, that is, $p[\vec{x}]$ is a subterm of ψ, Assume $p[\vec{x}]$ is selected as a pattern for ψ, and that a ground term g from E is such that $E \models_T g = p[\vec{t}]$. This furthermore implies that $E, \varphi[p[\vec{t}], \vec{t}] \models_T \varphi[g, \vec{t}]$, that

#Instances	**cvc3**		**cvc4**		**z3**	
1-10	1464	13.5%	1007	8.9%	1321	11.4%
10-100	1755	16.2%	1853	16.3%	2554	22.1%
100-1000	3816	35.2%	3680	32.4%	4553	39.4%
1000-10k	1893	17.4%	2468	21.7%	1779	15.4%
10k-100k	1162	10.7%	1414	12.5%	823	7.1%
100k-1M	560	5.2%	607	5.3%	376	3.3%
1M-10M	193	1.8%	330	2.9%	139	1.2%
>10M	10	0.1%	0	0.0%	8	0.1%

Figure 2: Number of instantiations by implementations of E-matching in SMT solvers on unsatisfiable benchmarks they solve with a 300 second timeout from SMT-LIB, TPTP and Isabelle. Data taken from [40].

is, by adding an instance of ψ to E we learn that $\varphi[g,\vec{t}]$ holds as well. In other words, the goal of E-matching is to, **from Q, learn information about ground terms in E**. In Example 3, the instantiation lemma returned by E-matching tells us that the predicate term $P(a,c)$ from E entails $g(b)=c$, under the assumption that E holds. This intuition will be revisited in Section 4.

3.1 Pattern Selection

There is no standard way to determine which patterns to select in E-matching. The performance of SMT solvers is highly dependent upon having a good method for pattern selection. In practice, patterns for quantified formulas can be can either be selected manually by the user, or selected automatically by the SMT solver. Recent work has explored more sophisticated pattern selection techniques [32].

A typical pattern selection for $\forall\vec{x}\,\varphi$ includes patterns that are subterms φ, and are applications of uninterpreted functions. For example, if φ is $f(x,y)=x+y$ where f is uninterpreted, then $f(x,y)$ would be selected as a pattern but not $x+y$. Moreover, they tend to select multiple eligible patterns when multiple are available, e.g. $P(x)$ and $R(x)$ may be selected when φ is $P(x)\vee R(x)$. Another key choice in pattern selection is whether to consider nested terms, e.g. when φ is $P(f(x))$, we may either select $f(x)$ or $P(f(x))$, where the latter selection leads to strictly fewer instantiations. If no eligible term contains all the free variables in $\vec{x}$, then it selects a pattern consisting of multiple terms. These are often called *multi-patterns*. For example, for $R(x,y)\wedge R(y,z)\Rightarrow R(x,z)$, then the set $\{R(x,y),R(y,z)\}$ is a multi-pattern. E-matching adds an instance for this multi-pattern if it could find a substitution for x,y,z that simultaneously matched both of these terms to ground terms in the current context. Multi-patterns tend to lead to many instantiations, and hence they tend to be chosen with lower priority. It is yet to be explored how recent techniques for literal selection in automated theorem provers [25] relate to pattern selection in SMT solvers.

3.2 Challenge: Too Many Instances

One of the central challenges when using E-matching is dealing with the large number of instantiations it generates. Figure 2 gives statistics on the number of instantiations returned by three implementations of E-matching in recent SMT solvers [6, 4, 15] on benchmarks they solve from the the SMT-LIB library [5], the TPTP library [47], and the Isabelle proof assistant [9]. For example, cvc4 solves 1853 benchmarks for which it added between 10 and 100 instantiation lemmas. We can see from this data that the median number of instantiations tends to be between 100 and 1000, and a majority of benchmarks taking between 10 and 10 thousand instantiations. A number of benchmarks are solved after upwards of one million instantiations, and z3 solves one benchmark in the TPTP library after adding more than 19.5 million instantiation lemmas.

The number of instantiation lemmas tends to increase exponentially over the course of a run of an SMT solver. The reason is that instantiation lemmas introduce new ground terms, which are in turn used to generate new substitutions that lead to new instantiation lemmas and so on. In practice, it is common for E-matching to generate thousands of instantiations after executing only a handful of iterations, which in turn overload the quantifier-free component of the SMT solver (Lines 2-6 of Figure 1), thereby degrading the performance of the system significantly. It is also important to note that E-matching may diverge on simple examples as demonstrated in the following example.

Example 4. *Let E be the set $\{f(a) = a\}$ and let ψ be $\forall x\, f(f(x)) = f(x)$. Say we select $f(x)$ as a pattern for ψ. Matching this pattern with $f(a)$ gives us the lemma $\psi \Rightarrow f(f(a)) = f(a)$. Adding the right hand side to E, we have that $f(x)$ matches $f(f(a))$, giving us the lemma $\psi \Rightarrow f(f(f(a))) = f(f(a))$. In absence of further heuristics, lemmas of this form will be added indefinitely.* □

The aforementioned case is often referred to as a *matching loop*. Matching loops may be avoided by restricting pattern selection, or by considering instantiations in a breadth-first manner by tracking a "level" on the terms introduced by instantiations [23].

3.3 Challenge: Incompleteness

In addition to being non-terminating, if E-matching terminates without return any instantiation lemmas, this does not imply that $E \cup Q$ is satisfiable, as demonstrated in the following simple example.

Example 5. *Let E be $\emptyset$ and let $Q = \{\forall x\, P(x), \forall x\, \neg P(x)\}$. Any pattern selection for the quantified formulas in Q will fail to produce any instantiation lemmas, since there are no ground terms in E. However, Q clearly is T-unsatisfiable.* □

Thus, when E-matching saturates with no instances, some SMT solvers will terminate with "unknown" or revert to more aggressive techniques for finding instantiations. Some variants of E-matching ensure completeness by a particular pattern selection specialized to the quantified formulas in question [18, 2]. However, these approaches require an external pencil-and-paper proof by the user and are not fully automated.

4 Conflict-Based Instantiation

As mentioned, E-matching often generates an overabundance of instantiation lemmas, many of which are irrelevant to the satisfiability of our input, and which degrade the performance of the SMT solver. To address this shortcoming, we may use a technique which we refer to as *conflict-based quantifier instantiation*. This technique was introduced in the context of a modern SMT solvers in [40], and has been implemented in solvers such as CVC4 and veriT [3].

Given a quantified formula $\forall \vec{x}\, \varphi[\vec{x}]$ in Q, we say that $\varphi[\vec{t}]$ is a *conflicting instance* for (E, Q) if $E, \varphi[\vec{t}] \models_T \bot$. A single conflicting instance exists for (E, Q) suffices to show that $E \cup Q$ is unsatisfiable, and hence we may return *only* that instance.

Example 6. *Let E be the set $\{P(a), \neg P(d), \neg P(c), \neg R(b), \neg R(a), \neg R(d)\}$ where a, b, c, d are free constants and P, R are unary predicates. Let ψ be $\forall x\, P(x) \vee R(x)$. Recall that E-matching would return instances corresponding to substituting terms a, b, c, d for x assuming $P(x)$ and $R(x)$ are chosen as* *patterns. However, consider the result of what we learn from these instantiations relative to information we already know from E:*

$$\begin{array}{rcl@{\qquad}rcl} E, P(a) \vee R(a) & \models & \top & E, P(b) \vee R(b) & \models & P(b) \\ E, P(c) \vee R(c) & \models & R(c) & E, P(d) \vee R(d) & \models & \bot \end{array}$$

In other words, since $P(a) \in E$, we learn no new information from the first instance, whereas since $\neg P(d), \neg R(d) \in E$, we learn $\bot$ from the fourth instance. In this example, $P(d) \vee R(d)$ is a conflicting instance for $(E, \{\psi\})$. Conflict-based instantiation would return only *this lemma and discard the others.* $\square$

Example 7. *We may find instantiations that are conflicting while taking into account equality and uninterpreted functions. Let E be $\{a \neq c, f(b) = b, g(b) = a, f(a) = a, h(f(a)) = d, h(b) = c\}$ and let ψ be $\forall x\, f(g(x)) = h(f(x))$. The instance $f(g(b)) = h(f(b))$ is a conflicting instance for $(E, \{\psi\})$, by noting $E \models_{\text{EUF}} f(g(b)) = a \wedge h(f(b)) = c \wedge a \neq c$ and hence $E, f(g(b)) = h(f(b)) \models_{\text{EUF}} \bot$.* $\square$

It is also helpful to distinguish other instances that are relevant to the satisfiability of $E \cup Q$, by noting which ones entail equalities between ground terms in E. This behavior is analogous to that of quantifier-free theory solvers, which often use theory propagation techniques for inferring equalities that prune search space.

Example 8. *Consider the sets E and ψ from the previous example, but where $a \neq c$ does not occur in E. In this case, $f(g(b)) = h(f(b))$ is no longer a conflicting instance for $(E, \{\psi\})$. However, notice that we still have that $E, f(g(b)) = h(f(b)) \models_{\text{EUF}} a = c$. In the context of Figure 1, adding this instance allows the quantifier-free solver to propagate the equality $a = c$. In this case, we say that $f(g(b)) = h(f(b))$ is a* propagating instance *for $(E, \{\psi\})$.* $\square$

To summarize, the goal of conflict-based instantiation is to **from Q, learn conflicts and equalities between ground terms in E**. We run conflict-based instantiation prior to running E-matching. A revised strategy for quantifier instantiation is the following:

1. If there exists a conflicting instance $\varphi[\vec{t}]$ for (E, Q), return $\{\forall \vec{x}\, \varphi[\vec{x}] \Rightarrow \varphi[\vec{t}]\}$ only,
2. Otherwise, if there exists a non-empty set of propagating instances $\{\varphi_1[\vec{t_1}], \ldots \varphi_n[\vec{t_n}]\}$ for (E, Q), return $\{\forall \vec{x}_1\, \varphi_1[\vec{x}_1] \Rightarrow \varphi_1[\vec{t_1}], \ldots, \forall \vec{x}_n\, \varphi_n[\vec{x}_n] \Rightarrow \varphi_n[\vec{t_n}]\}$ only.
3. Otherwise, return the instantiation lemmas returned by E-matching.

It is important to note that the level of effort put towards finding conflicting and propagating instances is flexible. An implementation that spends more time searching for conflicting instances will potentially add fewer instantiation lemmas, but will spend more time per lemma added.

4.1 Impact

Figure 3 gives a detailed summary of the impact of conflict-based instantiation in the SMT solver CVC4 on the SMT-LIB, TPTP, and Isabelle benchmark libraries (further details are given in [40]). The data shows CVC4 with and without conflict-based instantiation, indicated by **cvc4+ci** and **cvc4** respectively. Column 3 shows that CVC4 with conflict-based instantiation solves a total of 807 more benchmarks total over the three benchmark sets. Moreover, it uses significantly fewer instantiations overall to solve these benchmarks. Column 4 gives the aggregate number of instantiation rounds (IR) taken by these configurations (in terms of Figure 1, the number of instantiation rounds is equal to the number of calls to $\mathsf{check}_{\forall}$). While **cvc4+ci** considers many more instantiation rounds, it adds a total of 847.1 million fewer instantiations than **cvc4** over the three benchmark sets, as shown in column 5. In other words,

					E-matching		Conflicting		Propagating	
		# Solved	IR	# Inst	%IR	# Inst	%IR	# Inst	%IR	# Inst
TPTP	**cvc4**	6100	71.6K	879.0M	100.0	879.0M				
	cvc4+ci	6616	209.0K	150.9M	20.3	150.4M	76.4	159.7K	3.3	415.8K
Isabelle	**cvc4**	3858	7.0K	119.0M	100.0	119.0M				
	cvc4+ci	4082	21.8K	28.3M	22.4	28.2M	64.0	13.9K	13.6	130.9K
SMT-LIB	**cvc4**	3680	14.0K	60.7M	100.0	60.7M				
	cvc4+ci	3747	58.0K	32.4M	20.0	32.3M	71.6	41.5K	8.4	51.5K

Figure 3: Details on unsatisfiable benchmarks solved, and instances constructed by CVC4 with and without conflict-based instantiation.

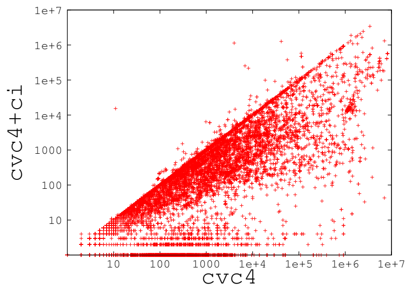


Figure 4: Number of instantiations reported by **cvc4+ci** vs **cvc4** over all unsatisfiable benchmarks. Data shown on a log-log scale.

conflict-based instantiation allows CVC4 to solve 807 more benchmarks while adding a total of 847.1 million less instantiations overall.

Columns 6-11 give the percentage of instantiation rounds taken by the solver for which it returns instantiations based on E-matching, a conflicting instance, or propagating instance(s). It is interesting to note that conflicting instances are found by **cvc4+ci** on 74.5% of instantiation rounds, a percentage that is relatively consistent across all three benchmark sets. When conflicting instances are not found, propagating instances are found on an additional 5.1% of instantiation rounds. A vast majority of the instantiations added by **cvc4+ci** are added as a result of E-matching (>99%).

Figure 4 gives a scatter plot of the number of instantiations per individual benchmark across all three sets, where the data is shown on a log scale. With the exception of only a few outliers, **cvc4+ci** considers fewer instantiations, and in many cases only requires a handful of instantiations to solve benchmarks for which **cvc4** requires thousands of instantiations. Results from [40] showed that 2520 benchmarks can be solved by **cvc4+ci** without resorting to E-matching at all. Moreover, 94 of these 2520 benchmarks could not be solved by **cvc4** within the timeout, showing a number of difficult benchmarks can be solved by conflict-based instantiation alone.

4.2 Challenge: Finding Conflicting Instances

A key challenge behind conflict-based instantiation is efficiently identifying which instantiations are conflicting. A naive approach would be to construct the set of instantiations that E-matching would return, and then check if each one of these instances was conflicting. This is highly inefficient, and partially negates our original motivation. Instead, implementation of conflict-based instantiation in CVC4 uses a stronger version of matching that takes into account the structure of the bodies of quantified formulas. As a simple example, for the formula $\forall x\, P(x) \vee R(x)$, we would match $P(x)$ against ground terms from E that are currently entailed to be equal to $\bot$, but not $\top$. Another challenge is finding conflicts in the presence of interpreted symbols.

Example 9. *Let E be the set $\{f(1) = 5\}$ and let ψ be $\forall xy\, f(x + y) > x + 2 \cdot y$. The instance $f(-3+4) > -3+2\cdot 4$ is conflicting for $(E, \{\psi\})$, noting that $E, f(-3+4) > -3+2\cdot 4 \models_{\text{UFLIA}} \bot$, where* UFLIA *is the combined theory of uninterpreted functions and linear integer arithmetic.* □

In general, finding conflicting instances can be expensive, both because checking entailment modulo theories may be time consuming, and because finding conflicting instances in the presence of background theories is often not obvious (as demonstrated in Example 9). For these reasons, the implementation of conflict-based instantiation in CVC4 considers entailment only modulo equality and uninterpreted functions, and avoids exponential behavior at the cost of being more incomplete for finding conflicting instances.

5 Model-Based Instantiation

We have seen that E-matching is an incomplete procedure, that is, when it terminates with no instantiation lemmas, this does not imply that our input is satisfiable. *Model-based instantiation* is technique that addresses this shortcoming, which was introduced in the context of modern SMT solving in [24], and has since been implemented in several systems [27, 42].

With model-based quantifier instantiation, the basic idea is to construct a *candidate model* $\mathcal{I}$ that satisfies E, and then check whether $\mathcal{I}$ satisfies Q as well. In other words, $\mathcal{I}$ contains a complete interpretation for all free constants and uninterpreted functions occurring in E and Q. Constructing an interpretation that satisfies the quantifier-free constraints in E is typically guaranteed since decision procedures for quantifier-free inputs in SMT tend to produce models. To check whether an interpretation $\mathcal{I}$ satisfies a quantified formula $\forall \vec{x}\, \varphi[\vec{x}]$ where φ is quantifier-free, it suffices to show that the quantifier-free formula $\neg\varphi^{\mathcal{I}}[\vec{k}]$ is *unsatisfiable*, where $\varphi^{\mathcal{I}}$ is the result of replacing each uninterpreted function in φ by its interpretation in $\mathcal{I}$. If $\neg\varphi^{\mathcal{I}}[\vec{k}]$ is *satisfiable*, we return an instantiation lemma based on the interpretation of k in a model for this formula.

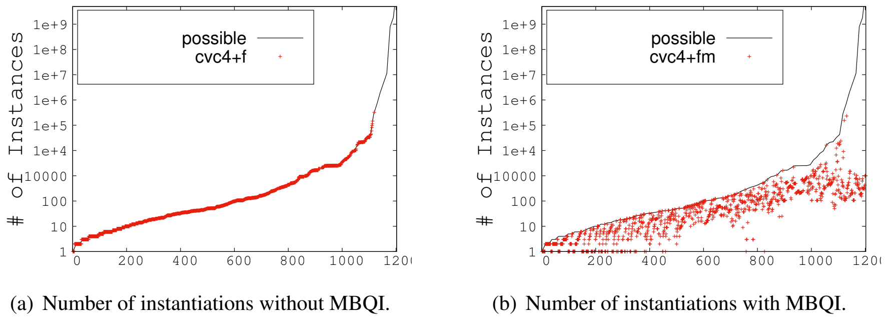
Example 10. *Let E be $\{\neg P(a), P(b), \neg R(b), \neg R(c)\}$ and let ψ be $\forall x\, P(x) \vee R(x)$. We may construct an interpretation $\mathcal{I}$ that satisfies E:*

$$\begin{array}{lllllllllll} P^{\mathcal{I}} & = & \lambda x. & \mathsf{ite}(& x = a, & \bot, & \mathsf{ite}(& x = b, & \top, & \top &)) \\ R^{\mathcal{I}} & = & \lambda x. & \mathsf{ite}(& x = b, & \bot, & \mathsf{ite}(& x = c, & \bot, & \bot &)) \end{array}$$

Notice that $P^{\mathcal{I}}$ was constructed as an if-then-else term that has entries corresponding to $P(a) \Leftrightarrow \bot$ and $P(b) \Leftrightarrow \top$. Its default *value (the rightmost occurrence of $\top$ in $P^{\mathcal{I}}$) was chosen arbitrarily. Similarly, we fix the existing entries and chose a default value of $\bot$ for R. To check whether $\mathcal{I}$ satisfies ψ, we check the satisfiability of:*

$$\neg(\forall x\, P^{\mathcal{I}}(x) \vee R^{\mathcal{I}}(x)) \rightsquigarrow \neg(P^{\mathcal{I}}(k) \vee R^{\mathcal{I}}(k)) \rightsquigarrow \neg(ite(k = a, \bot, \top) \vee \bot) \rightsquigarrow (k = a)$$

where k is fresh constant. In other words, after simplification, we find that ψ is not satisfied by $\mathcal{I}$ for all values of k such that $k = a$. Model-based quantifier instantiation would subsequently return the instantiation lemma $\psi \Rightarrow P(a) \vee R(a)$. Notice that this instantiation lemma is not satisfied by $\mathcal{I}$. Hence, the solver will be forced to choose a new candidate model if model-based quantifier instantiation is invoked again. □



(a) Number of instantiations without MBQI. (b) Number of instantiations with MBQI.

Figure 5: Number of instantiations added by CVC4 with and without model-based quantifier instantiation on satisfiable TPTP benchmarks with a 30 second timeout.

Example 11. *In the previous example, if we had chosen $\top$ as a default value for $R^{\mathcal{I}}$, our check would have:*

$$\neg(P^{\mathcal{I}}(k) \vee R^{\mathcal{I}}(k)) \rightsquigarrow \neg(ite(k=a,\bot,\top) \vee ite(k=b,\bot,ite(k=c,\bot,\top))) \rightsquigarrow \bot$$

This formula is unsatisfiable, thus, we cannot find a value of k for which the negation of $\forall x\, P(x) \vee R(x)$ holds in $\mathcal{I}$. In other words, $\mathcal{I}$ satisfies $\forall x\, P(x) \vee R(x)$. □

Intuitively, model-based quantifier instantiation adds lemmas that refine candidate models until they are also a model of Q, or otherwise the SMT solver finds a conflict.

5.1 Impact

Model-based instantiation often enables SMT solvers to establish the satisfiability of inputs where exhaustive instantiation is infeasible. In [42], an implementation of model-based quantifier instantiation is used in conjunction with finite model finding techniques in CVC4. Figure 5 shows the impact of these techniques for the 1202 benchmarks from the TPTP library. In both plots, the solid line plots the total number of possible instances of all quantified formulas in the (smallest) finite model found by CVC4. For instance, the quantified formula $\forall xyz\, P(x,y,z)$ has $2^3 = 8$ possible instantiations in a model that interprets the domain of x,y,z each as a set of size 2.

The left plot of Figure 5 shows the number of instances added by a strategy **cvc4+f** that exhaustively instantiates all quantified formulas, where this number coincides with the number of possible instantiations for benchmarks for which **cvc4+f** terminates with a model. This plot shows that **cvc4+f** scales up to benchmarks having only around 100k instances; the maximum number of instances considered by **cvc4+f** on a benchmark it solved was 323k. In contrast, the right plot of Figure 5 shows the number of instances added with model-based quantifier instantiation (**cvc4+fm**). This strategy solves a superset of the benchmarks solved by **cvc4+f**, solving some benchmarks having more than 1 billion possible instantiations, and often adding far fewer than the number of possible instantiations before terminating with a model as demonstrated by the points in the plot occurring below the solid line.

5.2 Completeness

We have now seen that SMT solvers may answer "unsat" using quantifier instantiation when they discover a finite set of instances that are collectively unsatisfiable at the quantifier free, and may answer "sat" if they discover a candidate model that satisfies all quantified formulas. It is worth noting when these solvers are guaranteed to terminate.

Approaches that incorporate model-based instantiation are clearly terminating when the domains of quantified formulas have a fixed finite interpretation such in quantified Boolean formulas (QBF) [28], or quantified bit-vectors [49]. For quantified formulas over unintepreted sorts, finite model finding approaches [41] are generally finite-model complete, that is guaranteed to terminate with "sat" when there exists a model that interprets all uninterpreted sorts as finite sets, and are complete for fragments that exhibit a small model property. More generally, approaches for model-based quantifier instantiation are terminating in cases where it can be argued that only a finite number of instantiation lemmas will be returned before a model is found [26, 24]. The work of [24] examines a class of formulas known as the essentially uninterpreted fragment, where it can be argued that if a computation of the relevant domain of quantified formulas is finite, then an instantiation strategy can be made that is complete. This fragment includes some quantified formulas over infinite domains such as the integers.

5.3 Challenge: Constructing Models

The core challenge to model-based quantifier instantiation is how to construct candidate models $\mathcal{I}$. As mentioned, common approaches to model-based quantifier instantiation construct almost constant interpretations for all uninterpreted functions. This limits their strength both in terms of performance in practice and completeness.

Example 12. *Let ψ be $\forall xy\, f(x,y) \geq x \wedge f(x,y) \geq y$, where x and y are integers. This formula has a model where f is interpreted as $\lambda xy\, \mathsf{ite}(x \geq y, x, y)$, but does not have a model where f is almost constant.* $\square$

Example 13. *Let ψ be $\forall x\, 3\cdot g(x)+5\cdot h(x) = 0$. This formula has a model where g and h are interpreted as $\lambda x\, 5 \cdot x$ and λx -3 $\cdot\, x$ respectively, but does not have a model where they are almost constant.* $\square$

6 Specialized Instantiation for Theories

We have seen three classes of general-purpose techniques for instantiation for quantified formulas in SMT. The common thread thus far has been that determining the satisfiability of universally quantified formulas in presence of uninterpreted functions and theory symbols is challenging, and in fact is undecidable in general. However, when no uninterpreted function symbols are present and the background theory admits quantifier elimination, e.g. linear real and integer arithmetic, then satisfiability can be established by a quantifier elimination procedure [13, 21, 33]. Recent work has adapted quantifier elimination techniques in the context of SMT solving [34, 38, 8], and has been implemented in SMT solvers as well as tools that use SMT solvers as a backend [29, 20].

Let $\exists\vec{x}\,\varphi[\vec{x},\vec{k}]$ be a quantified formula in a background theory T. Assuming T admits quantifier elimination, we know that this formula is equivalent to some finite disjunction $\varphi[\vec{t_1},\vec{k}] \vee \ldots \vee \varphi[\vec{t_n},\vec{k}]$ where $\vec{t_1},\ldots,\vec{t_n}$ are tuples of terms, possibly containing free constants from $\vec{k}$. We may alternatively consider a lazy quantifier *instantiation* procedure for establishing the satisfiability $\forall\vec{x}\,\neg\varphi[\vec{x},\vec{k}]$ that considers only instantiations of the form $\neg\varphi[\vec{t_i},\vec{k}]$ where $\vec{t_i}$ is one of $t_1,\ldots,t_n$. Assuming instantiations of this form are considered by the procedure only, it is terminating. Moreover, the procedure may terminate early if at any point it finds a set $\{\neg\varphi[\vec{t_{i1}},\vec{k}],\ldots,\neg\varphi[\vec{t_{im}},\vec{k}]\}$ that is either T-unsatisfiable, or is T-satisfiable and entails $\forall\vec{x}\,\neg\varphi[\vec{x},\vec{k}]$, where ideally m is much smaller than n.

An approach for selecting instantiation lemmas based on models for the negation of quantified formulas over linear arithmetic is given in [39], where the basic idea is the following. Let ψ be a quantified formula of the form $\forall x\, \varphi[x]$ where φ is a quantifier-free formula over linear real arithmetic. First, find a model for $E \cup \neg\varphi[k]$ where E is the current set of quantifier-free constraints known by the solver and k is fresh real variable. If none exists, then ψ is satisfied by all models of E. Otherwise, let $\{k \geq t_1, \ldots, k \geq t_n\}$ be (equivalent to) atoms from $E \cup \neg\varphi[k]$ that are satisfied by $\mathcal{I}$, find the bound $k \geq t_i$ such that $t_i^{\mathcal{I}} \geq t_j^{\mathcal{I}}$ for all $j \neq i$, and return the instantiation lemma $\psi \Rightarrow \varphi[t_i]$. In other words, select an instantiations based on the current maximal lower bound for k (alternatively, instantiations may be selected based on the current minimal upper bound).

Example 14. *Let E be the set $\{a = b+5\}$ and let ψ be $\forall x\,(x > a \vee x < b \vee x - c < 3)$. Consider the set $E' = E \cup \{k \leq a, k \geq b, k \geq c+3\}$ where k is a fresh variable. This set is satisfiable with a model $\mathcal{I}$ that interprets $a^{\mathcal{I}} = 5, b^{\mathcal{I}} = 0, c^{\mathcal{I}} = 0, k^{\mathcal{I}} = 3$. Consider the lower bounds for k in set E', and their interpretation in $\mathcal{I}$:*

$$\begin{array}{ccccc} k & \geq & b^{\mathcal{I}} & = & 0 \\ k & \geq & (c+3)^{\mathcal{I}} & = & 3 \end{array}$$

We find the term $c+3$ is the maximal lower bound for k in $\mathcal{I}$, and return the instantiation lemma $\psi \Rightarrow (c+3 > a \vee c+3 < b \vee (c+3) - c < 3)$. Note that any model $\mathcal{J}$ satisfying E' and this instantiation lemma is such that $b^{\mathcal{J}} > (c+3)^{\mathcal{J}}$, and hence $c+3$ will not be the maximal lower bound for k for such a $\mathcal{J}$. □

In practice, The aforementioned approach for instantiation for linear arithmetic often terminates well before considering the worst-case number of instantiations [8, 39]. Here, we have shown one option for selecting instantiations based on maximal lower or minimal upper bounds, which intuitively can be understood as a lazily enumerating the disjuncts in Loos and Weispfenning's method for quantifier elimination [33]. Although not shown here, the instantiations returned by this approach may involve *virtual terms* such as infinitesimals for dealing with cases where bounds on k are strict, and infinities for dealing with cases where k is unbounded. Another alternative for linear arithmetic is to instantiate with the midpoint of the maximal lower and minimal upper bounds, which can be understood as enumerating disjuncts in Ferrante and Rackoff's method for quantifier elimination [21]. An instantiation strategy for linear integer arithmetic can be devised that adds constants to lower bounds to take into account divisibility constraints (for details, see [39]), which simulates Cooper's method [13]. For other background theories such as fixed-width bit-vectors, a naive approach can be devised that instantiates the original quantified formula based on the *value* $k^{\mathcal{I}}$. In all of the aforementioned cases, we may argue that the set of possible instantiation lemmas returned by this strategy is finite. Hence, a terminating procedure for the satisfiability of quantified formulas in these theories based on lazy quantifier instantiation can be devised.

7 Conclusion

We have seen strategies for quantifier instantiation based on heuristics, conflicts, and models. For quantified formula with uninterpreted functions, a strategy for quantifier instantiation may give highest priority to instantiations that are conflicting with the current satisfying assignment, subsequently give priority to instantiations returned by heuristics such as E-matching, and finally consider instances that refine candidate models. Quantified formulas in pure background theories such as linear arithmetic can be decided by particular strategies for selecting instantiation lemmas, for instance based on candidate models and the current set of constraints known by the solver.

There is room to improve upon current implementations of quantifier instantiation in SMT solvers, both in terms of engineering and theory. There are many design decisions behind all strategies mentioned in this paper, many of which are not mentioned here. It has yet to be explored how conflict-based instantiation can be implemented eagerly, and if it can be efficiently applied in the presence of background theories. There is also potential for incorporating more sophisticated schemes for function models that go beyond almost constant functions for model-based quantifier instantiation.

As mentioned, quantifier instantiation techniques based on E-matching are often highly effective and well-behaved for fragments where theory symbols are scarce. Conversely, recent techniques for quantifier instantiation are highly effective for quantifiers containing purely theory symbols, as mentioned in Section 6. For tackling problems that depend both on reasoning about theories and uninterpreted functions, it is yet to fully understood to what extent these techniques can be combined, and how this combination can be implemented in practice.

Additionally, there are opportunities to extend SMT solvers with efficient dedicated support for new fragments of first order logic of interest, including pure quantified formulas in additional background theories. There is potential for improving support for quantified formulas over fixed-width bit-vectors, where the bit-precise versions of arithmetic operators are part of the signature supported by many SMT solvers. We expect analogous techniques for selecting instantiations in the spirit of Section 6 to be helpful here. It has further yet to be explored how instantiation techniques can be adapted for other theories such as unbounded character strings, finite sets, and floating points, which several SMT solvers now support at the quantifier-free level.

References

[1] E. Althaus, E. Kruglov, and C. Weidenbach. Superposition modulo linear arithmetic SUP(LA). In *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, pages 84–99, 2009.

[2] K. Bansal, A. Reynolds, T. King, C. Barrett, and T. Wies. Deciding local theory extensions via e-matching. In *Computer Aided Verification (CAV)*. Springer, 2015.

[3] H. Barbosa. Efficient instantiation techniques in SMT (work in progress). In *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning, Coimbra, Portugal, July 2nd, 2016.*, pages 1–10, 2016.

[4] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification (CAV)*. Springer, 2011.

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

[6] C. Barrett and C. Tinelli. CVC3. In *CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 298–302, 2007.

[7] P. Baumgartner, J. Bax, and U. Waldmann. Beagle - A hierarchic superposition theorem prover. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 367–377, 2015.

[8] N. Bjørner and M. Janota. Playing with quantified satisfaction. In *LPAR 2015, Suva, Fiji, November 24-28, 2015.*, pages 15–27, 2015.

[9] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

[10] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, et al. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[11] T. Bouton, D. C. B. D. Oliveira, D. Déharbe, and P. Fontaine. verit: An open, trustable and efficient smt-solver. In *CADE-22, Montreal, Canada, August 2-7, 2009. Proceedings*, pages 151–156, 2009.

[12] R. Chapman and F. Schanda. Are we there yet? 20 years of industrial theorem proving with SPARK. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 17–26, 2014.

[13] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence, pages 91100*, 1972.

[14] L. de Moura and N. Bjørner. Engineering dpll (t)+ saturation. In *International Joint Conference on Automated Reasoning*, pages 475–490. Springer Berlin Heidelberg, 2008.

[15] 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*, pages 337–340. Springer-Verlag, 2008.

[16] L. M. de Moura and N. Bjørner. Efficient e-matching for SMT solvers. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

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

[18] C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Adding decision procedures to smt solvers using axioms with triggers. *Journal of Automated Reasoning*, 56(4):387–457, 2016.

[19] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2), 2006.

[20] G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Automated discovery of simulation between programs. In *LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, pages 606–621, 2015.

[21] J. Ferrante and C. W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer, 1979.

[22] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 55–64. IEEE, 2003.

[23] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, volume 4603 of *LNCS*. Springer, 2007.

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

[25] K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the selection. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 313–329, 2016.

[26] 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.

[27] S. Jacobs. Incremental instance generation in local reasoning. In *CAV '09*, pages 368–382, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] M. Janota, W. Klieber, J. Marques-Silva, and E. Clarke. Solving qbf with counterexample guided refinement. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–128. Springer Berlin Heidelberg, 2012.

[29] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification*. Springer International Publishing, 2014.

[30] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[31] K. R. M. Leino. Developing verified programs with dafny. In *ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1488–1490, 2013.

[32] K. R. M. Leino and C. Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 361–381, 2016.

[33] R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.

[34] D. Monniaux. Quantifier elimination by lazy model enumeration. In *Computer Aided Verification*, pages 585–599. Springer Berlin Heidelberg, 2010.

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

[36] 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 (JACM)*, 53(6):937–977, 2006.

[37] G. Reger, M. Suda, and A. Voronkov. Playing with AVATAR. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 399–415, 2015.

[38] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 198–216, 2015.

[39] A. Reynolds, T. King, and V. Kuncak. Solving quantified linear arithmetic by counterexample-guided instatiation (in submission). In *Formal Methods in System Design*, 2016.

[40] A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD*, pages 195–202. IEEE, 2014.

[41] 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.

[42] A. Reynolds, C. Tinelli, A. Goel, S. Krstic, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *CADE-24, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013.

[43] P. Rümmer. E-matching with free variables. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, pages 359–374, 2012.

[44] S. Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3):111–126, 2002.

[45] S. Schulz and M. Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 330–345, 2016.

[46] A. Stump, C. W. Barrett, and D. L. Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification (CAV)*, pages 500–504. Springer Berlin Heidelberg, 2002.

[47] 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.

[48] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In *International Conference on Automated Deduction*, pages 140–145. Springer Berlin Heidelberg, 2009.

[49] C. M. Wintersteiger, Y. Hamadi, and L. De Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.