Incremental Search for Conflict and Unit Instances of Quantified Formulas with E-Matching

Jochen Hoenicke(✉) and Tanja Schindler(✉)

University of Freiburg, Freiburg im Breisgau, Germany
{hoenicke,schindle}@informatik.uni-freiburg.de

Abstract. We present a new method to find conflicting instances of quantified formulas in the context of SMT solving. Our method splits the search for such instances in two parts. In the first part, E-matching is used to find candidate instances of the quantified formulas. In principle, any existing incremental E-matching technique can be used. The incrementality avoids duplicating work for each small change of the E-graph. Together with the candidate instance, E-matching also provides an existing node in the E-graph corresponding to each term in this instance. In the second part, these nodes are used to evaluate the candidate instance, i.e., without creating new terms. The evaluation can be done in constant time per instance. Our method detects conflicting instances and unit-propagating instances (clauses that propagate new literals). This makes our method suitable for a tight integration with the DPLL($\mathcal{T}$) framework, very much in the style of an additional theory solver.

1 Introduction

Satisfiability Modulo Theories (SMT) solving is the problem of finding solutions for first-order formulas or proving unsatisfiability and has many applications, e. g., in software verification, scheduling, program synthesis. Many SMT solvers are based on the DPLL($\mathcal{T}$) framework, where a DPLL engine assigns truth values to ground literals, thereby creating a partial model. Specialized solver modules for each theory check the feasibility of the model or report conflicting literal assignments or new facts (ground literals) that are implied by the theory. Usually, these theory solvers handle only the quantifier-free fragment of the corresponding theory. A common approach to deal with quantified formulas is to add instances of the quantified formulas to the ground part of the problem in order to prove unsatisfiability. A challenge is to select those instances that are useful for the solving process, as adding too many formulas overloads the solver. Finding the most promising instances is an active topic of research [1,4,5,8,13–15].

In the context of the DPLL($\mathcal{T}$) framework, a conflicting instance that refutes the partial model provided by the DPLL engine is most useful [15]. Other useful

Partially supported by the German Research Council (DFG) under HO 5606/1-2.

https://doi.org/10.1007/978-3-030-67067-2_24

instances are unit-propagating instances that show that a literal is implied by the partial model and allow the solver to assign the literal correctly. We present a new method for finding such conflicting or unit-propagating instances on the fly as the DPLL engine builds the partial model. This enables a tight integration of quantifier reasoning with the DPLL($\mathcal{T}$) framework.

The basic idea in DPLL($\mathcal{T}$) based solvers is to separate Boolean reasoning from theory reasoning. A DPLL engine searches for a solution of the Boolean core of the formula by guessing literals that should be true and propagating consequences from these guesses. The theory solvers guide the search by constantly checking if there is a model in the corresponding theory for the partial Boolean solution. If a theory solver finds a conflict, i.e., a subset of literals that together are unsatisfiable in the theory, then this is immediately reported to the DPLL engine in form of a lemma that states that one of the literals must be false. The DPLL engine backtracks decisions that lead to the conflict, and continues the search for a solution of the Boolean core augmented with the new lemma. This allows the DPLL engine to skip huge parts of the search space. Moreover, theory solvers can provide unit clauses that show that a literal must be true in the current context. This also reduces the search space considered by the DPLL engine, and has been shown to be effective for several quantifier-free logics [11].

We think that this approach is applicable for quantifier reasoning for the same reasons. Instead of adding many instances at a time, we consider a quantifier solver as one of the theory solvers in the DPLL($\mathcal{T}$) framework. That is, the quantifier solver actively participates in the search for a satisfying solution of a given problem by providing useful instances that guide the search in the right direction. A useful instance can be a conflicting instance that shows that the search took a wrong branch, or a unit-propagating instance that propagates a new fact. The core of our method is an incremental search for such conflicting and unit instances, that uses an incremental E-matching module. The incrementality is essential for the quantifier solver to find new instances without repeating the full search after each step in the solving process.

E-matching is the problem of finding ground terms that match a so-called *pattern*, i.e., a term that may contain variables. A term matches a pattern if it is equal, up to congruence, to the pattern instantiated with a suitable variable substitution. E-matching is used in many existing solvers as a heuristic to find potentially useful instances. The idea is to choose a set of patterns (a *multi-pattern*) for a quantified formula such that all variables are contained. An instance of a quantified formula is considered to be relevant if all patterns match for a common variable substitution. The success of E-matching based instantiation is strongly influenced by the choice of patterns. If the patterns are too restrictive, a relevant instance may not be found; if the patterns are too general, many irrelevant instances may be produced.

To illustrate E-matching based instantiation and its shortcoming, we consider the following example formula.

$$f(a,b) = a \wedge f(b,b) = b \wedge f(b,c) = c \wedge a = c \wedge b \neq c$$
$$\wedge \; (\; \forall x,y,z. \; f(x,y) \neq c \vee f(y,z) \neq c \vee f(x,z) = c \;)$$

A multi-pattern suitable for E-matching in the universally quantified subformula is $f(x,y), f(y,z)$. The E-matching engine matches each pattern with the terms in the ground part of the formula, to find values for x, y, and z, such that both instantiated patterns have an existing congruent term. One potential match yields the ground terms $f(b,c)$, $f(a,b)$ and the substitution $\{x \mapsto b, y \mapsto c, z \mapsto b\}$. This is a valid match: the instantiated second pattern $f(c,b)$ is congruent to $f(a,b)$ since $a = c$ is part of the ground formula. The instantiated clause $f(b,c) \neq c \vee f(c,b) \neq c \vee f(b,b) = c$ leads to a contradiction with the ground part and shows that the formula is unsatisfiable.

However, E-matching also finds a lot of instances that are not useful to show unsatisfiability. In the above example, also $\{x \mapsto a, y \mapsto b, z \mapsto b\}$ is matching the pattern. The corresponding instance is already satisfied as the last literal of the clause, $f(a,b) = c$, is already true. In total, E-matching finds five instances in this small example, of which three are already true, one is a conflict, and one derives some fact about the non-existing ground term $f(a,a)$. The main problem with producing irrelevant instances is that they can trigger new matches. This may even lead to so-called matching loops, e.g., if a new term from an instantiated formula matches the pattern again leading to increasingly larger variable substitutions.

E-matching is not only useful to find candidates for conflicting instances, it also provides congruent terms that can be used to evaluate the instances without any extra work. In the example above, $f(c,b) \neq c$ can be evaluated using the congruent term $f(a,b)$ for $f(c,b)$. This insight is the core of our method to find conflicting instances fast enough to be used as a DPLL($\mathcal{T}$) theory solver.

Our incremental search for conflicting and unit-propagating clauses is subdivided into two parts. First we search for candidate substitutions for quantified clauses by using E-matching for the quantified terms in the clause. We use the congruent terms provided by E-matching in the second part to evaluate the clause instance without actually building the instantiated terms. Only if the instance is found to be conflicting or unit-propagating, it is created. The approach of splitting this search into two parts has the advantage that the search for candidate substitutions using E-matching can be done incrementally [3] and does only little work each time a new ground literal is set or removed. The clause evaluation can be done literal by literal, which allows to not only detect conflicting instances, but also instances that propagate new literals.

We introduce the notation and basic definitions in Sect. 2. In Sect. 3, we give a brief overview of the DPLL($\mathcal{T}$) framework. We describe the congruence closure algorithm which is a decision procedure for the theory of equality, and outline E-matching based instantiation. In Sect. 4, we present our approach to find conflicting and unit-propagating instances of quantified formulas, and give theoretical results on correctness and completeness of the approach. Experimental evidence of the usefulness of our approach is given in Sect. 5. Finally, we mention related work in Sect. 6, and discuss future work in Sect. 7.

2 Notation and Basic Definitions

We assume standard sorted first-order logic with equality. A first-order *theory* is defined by its signature consisting of constant, function and predicate symbols, and a set of axioms for its interpreted symbols. We consider in the following mainly the theory of equality and uninterpreted functions $\mathcal{T}_E$. The axioms of $\mathcal{T}_E$ establish reflexivity, symmetry and transitivity for the equality symbol $=$, and congruence for each uninterpreted function symbol.

A *term* is a variable, a constant, or the application of an n-ary function to n terms. An *atom* is the application of an n-ary predicate to n terms. A *literal* is an atom or its negation. A *clause* is a disjunction of literals. A term, literal or clause is *ground* if it does not contain variables.

In the following, we assume w.l.o.g. that every formula is in conjunctive normal form (CNF), i.e., it is a conjunction of clauses. We also assume that every variable occurring in the formula is universally quantified. The latter can be established by introducing Skolem variables or functions for existentially quantified variables [12]. Thus, the formula is a conjunction of clauses and each clause implicitly universally quantifies over its free variables.

We use the letters a, b, c to denote constant symbols, the letters f, g, h to denote uninterpreted function symbols, and the letters x, y, z to denote universally quantified variables. We use the letter t to denote ground terms and the letter p to denote terms that may contain free variables (patterns). We use the letter ℓ for literals, F and φ for formulas, and C for clauses. We use the symbol $\bot$ to denote the formula that is always false. We write $p[x_1, \ldots, x_n]$, $\ell[x_1, \ldots, x_n]$, $\varphi[x_1, \ldots, x_n]$, and $C[x_1, \ldots, x_n]$ for terms, literals, formulas, and clauses, respectively, containing at most the variables $x_1, \ldots, x_n$. For a formula F, we write T_F to denote the set of all terms occurring in F.

A *substitution* is a mapping from variables to terms, and it is a *ground* substitution if it maps all variables to ground terms. We write $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for the substitution that maps variable x_i to term t_i for $i \in \{1, \ldots, n\}$. We also use the notation $p[x \mapsto t]$ to denote the term that results from replacing the variable x in p with the term t.

3 Preliminaries

In this section, we outline standard methods in SMT solvers that are the basis for our approach, namely the DPLL($\mathcal{T}$) framework that separates Boolean reasoning from theory reasoning in SMT solvers, the congruence closure algorithm which is an efficient decision procedure for the quantifier-free fragment of the theory of equality $\mathcal{T}_E$, and finally the technique of E-matching based instantiation which is a common approach to find useful instances of quantified formulas.

3.1 DPLL($\mathcal{T}$)

Many SMT solvers are based on the DPLL($\mathcal{T}$) framework. The basic idea is to separate Boolean reasoning from theory reasoning. The DPLL engine takes care of the propositional core of the CNF formula by assigning truth values to literals. In particular, it tries to satisfy each clause in the formula by assigning at least one literal to `true`. A clause where all literals are assigned to `false` is called a *conflict clause*. A clause where all but one literals are assigned to `false`, and this literal has not yet been assigned, is called a *unit clause*, and can be used to propagate this literal. If no unit clauses exist, the DPLL engine must make decisions on literals which may have to be backtracked if they lead to a conflict.

During the solving process, the currently assigned literals are passed to *theory solvers* that use specialized decision procedures. If a theory solver finds that the conjunction of literals is in conflict with the theory axioms, it returns a corresponding conflict clause (a subset of literals that are unsatisfiable) to the DPLL engine. The theory solver can also propagate literals that must be true in the theory under the current partial literal assignment by providing a unit clause. Conflict clauses may only contain existing literals, but theory solvers can create new literals that may be propagated by a unit clause to the DPLL engine or to other theories. In order to determine that a satisfying assignment has been found, a theory solver must also be able to provide a complete model.

The interaction between the DPLL engine and the theory solvers can happen in several stages. While it is enough to report any conflicts once all literals have been assigned, finding conflicts early and propagating literals implied by the theory during the search for a Boolean model can often help the DPLL engine to significantly reduce the search space [11]. However, theory reasoning comes with a certain cost, which is why it does not always make sense to compute all theory conflicts and propagations in each step of the solving process. Finding the right compromise between efficiency and completeness of theory propagation is the key in building an efficient solver.

3.2 Congruence Closure

The quantifier-free conjunctive fragment of the theory of equality $\mathcal{T}_E$ can be decided by computing the congruence closure for the equality relation on a graph representing the involved terms [10].

An *E-graph* is a graph with nodes (vertices) and two kind of edges. Figure 1 shows an example. Each node in the graph represents a term and for every term there is at most one node. If the term is a function application, it is labelled by the function symbol and solid edges point from the node to the arguments of the function application. Dashed edges, the so-called equality edges, represent equalities between these terms that were decided by the DPLL engine or that are propagated congruences. Let $\sim$ denote the transitive closure of all equality edges, i.e., $t_1 \sim t_2$ is true if and only if t_1 and t_2 are connected by a sequence of equalities. The connected components $[t]_= = \{t' \mid t \sim t'\}$ are called the congruence classes. There is an efficient algorithm based on union-find data

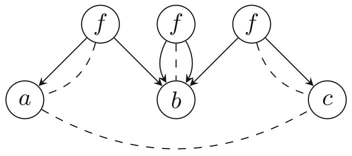

Fig. 1. E-graph for the formula $f(a, b) = a \wedge f(b, b) = b \wedge f(b, c) = c \wedge a = c$. Nodes represent terms, solid arrows between nodes symbolize function-argument relations, and dashed lines symbolize equality.

structures that builds the E-graph in $O(n \log n)$. The lookup whether $t_1 \sim t_2$ holds for two nodes t_1 and t_2 in the graph is in $O(1)$.

The congruence closure algorithm works incrementally in a DPLL setting. It starts with the empty E-graph that does not have any equality edge. When the DPLL engine decides or propagates an equality, the corresponding nodes are connected with an equality edge. If a disequality is decided, it is remembered for the corresponding pair of congruence classes (this information is updated whenever congruence classes are merged by a new equality edge).

Whenever congruence classes are merged, the congruence closure algorithm also propagates all implied *congruences*. For each pair of function application terms $f(t_1, \ldots, t_n)$ and $f(t'_1, \ldots, t'_n)$ on the same function symbol, the algorithm checks whether $t_i \sim t'_i$ holds for $1 \leq i \leq n$. If this is the case, the congruent function applications are connected by an equality edge. There are efficient data structures to quickly find the candidate application terms that may be affected by a previous merge of two congruence classes.

When a disequality is set between two terms with $t_1 \sim t_2$ or if two congruence classes are merged that already have a disequality between them, the congruence closure algorithm reports a conflict. This conflict can be explained by the disequality $t_1 \neq t_2$ and the path of equality edges between t_1 and t_2.

For terms t_1, t_2 existing as nodes in the E-graph, it can be determined in constant time (each node remembers its representative) whether the literals decided by the DPLL engine imply an equality $t_1 = t_2$. For literals $t_1 \neq t_2$, the algorithm can check if there a disequality set between the corresponding congruence classes. In that case the literals decided by the DPLL engine imply the disequality. However, not all implied disequalities can be found this way. For example, the literal $f(a, b) \neq f(b, a)$ implies $a \neq b$ but the congruence closure algorithm would not find this disequality.

3.3 E-Matching Based Instantiation

A common approach in SMT solvers to handle problems containing quantified formulas is to add instances of the quantified formulas to the ground part of the problem, and solve the resulting ground formula. A heuristic method to find instances that help the solving process is based on E-matching. It was first implemented in the Simplify theorem prover [5]. An incremental version has been presented for instance in [3].

E-matching is the problem of finding terms in the E-graph that match a given *pattern* (a term with free variables) up to congruence. The idea of E-matching based instantiation is that an instance $\varphi\sigma$ of a universally quantified formula $\forall x_1, \dots, x_n.\ \varphi[x_1, \dots, x_n]$ is useful to solve the problem if it contains enough terms that are congruent to terms in the current E-graph, as such an instance allows for deriving new information about existing terms. In order to find such instances, non-ground terms $p_1, \dots, p_n$, a so-called *multi-pattern* (or trigger), from the formula φ are selected. They should contain all free variables of φ in order to extract a substitution from a match. The E-matching algorithm then searches for terms $t_1, \dots, t_n$ in the E-graph, and a substitution σ, such that $t_1 \sim p_1\sigma, \dots, t_n \sim p_n\sigma$ holds, where $\sim$ denotes the congruence closure of the equality edges of the E-graph.[1] We say that each t_i matches the pattern p_i. Here, t_i and $p_i\sigma$ need not be the same term, but congruent terms. In particular, t_i occurs explicitly in the E-graph, while $p_i\sigma$ does not necessarily occur there. For $p_i = f(p'_1, \dots, p'_m)$, this means that the congruence class of t_i contains a term $f(t'_1, \dots, t'_m)$ such that $t'_j \sim p'_j\sigma$ holds.

Example 1. Let $F : f(a) = b \land g(b) = c \land \forall x.\ g(f(x)) = d$. A useful pattern for E-matching is $p : g(f(x))$. Figure 2 shows the E-graph for the ground part, and how the pattern p is matched. The result of applying E-matching is $\sigma = \{x \mapsto a\}$ and the term t with $t \sim p\sigma$ is $g(b)$. Note that $p\sigma = g(f(a))$ does not exist in the E-graph.

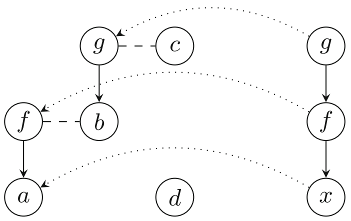

Fig. 2. E-matching for $F : f(a) = b \land g(b) = c \land \forall x.\ g(f(x)) = d$. The left part displays the E-graph for the ground part of F, the right part displays the pattern $g(f(x))$ for the quantified part. Solid arrows symbolize function-argument relations, dashed lines symbolize equality, and the dotted arrows display which terms in the E-graph match with which subterm of the pattern.

E-matching is usually used as a basis for a heuristic instantiation procedure. For a quantified formula φ and a corresponding multi-pattern $p_1, \dots, p_n$, whenever matching terms $t_1, \dots t_n$ and a substitution σ with $t_i \sim p_i\sigma$ are found, the instance $\varphi\sigma$ is added to the ground problem. One problem with E-matching is to

[1] From now on $\sim$ denotes the congruence closure of the equality edges in the E-graph and not the transitive closure as in the previous section. Note that this is also defined for $p\sigma$, if it does not exist in the E-graph.

choose the right multi-pattern. If the pattern is too strict, important instances may be missed. If the pattern is too loose, it creates too many irrelevant instances which may cause new matches. In general, E-matching cannot be used to show satisfiability for a quantified formula; although there are quantified formulas with patterns for which E-matching is complete.

4 Finding Conflict and Unit Instances

In this section we describe our algorithm to find conflict and unit instances for quantified clauses. We assume that the input formula was preprocessed into conjunctive normal form and that existential quantifiers were skolemized by introducing fresh constants or function symbols [12]. All variables occurring in clauses are universally quantified. Thus, a quantified clause always is of the form

$$\forall x_1, \ldots, x_n.\ \ell_1[x_1, \ldots, x_n] \vee \ldots \vee \ell_m[x_1, \ldots, x_n]$$

where $\ell_1, \ldots, \ell_m$ are literals containing at most the variables $x_1, \ldots, x_n$. We omit the universal quantifier and implicitly see all free variables in a clause as universally quantified.

The quantified clauses are handled by a separate quantifier theory. Given a quantified clause $C[x_1, \ldots, x_n]$, the theory searches for a ground substitution σ for $x_1, \ldots, x_n$ such that the resulting instance $C\sigma$ is in conflict with the current partial model, or leads to a propagation. We define such instances as follows.

Definition 1. *Let M be a (partial) literal assignment and let $C := \ell_1 \vee \ldots \vee \ell_m$ be the body of a quantified clause containing the free variables $x_1, \ldots, x_n$.*

1. *A ground instance $C\sigma$ for a substitution σ over $x_1 \ldots x_n$ is* conflicting *if $M \cup \{C\sigma\} \models_{\mathcal{T}} \bot$, or, equivalently, $M \models_{\mathcal{T}} \neg\ell_i\sigma$ for all $i \in \{1, \ldots, m\}$. (This definition follows [15].)*
2. *A ground instance $C\sigma$ is* unit-propagating *if there is an i such that $M \models_{\mathcal{T}} \neg\ell_j\sigma$ for all $j \neq i$.*

Note that this definition does not require a clause instance resulting from a conflicting or unit-propagating instance to be in conflict with the Boolean model, i.e., it is not a conflict or unit clause for the DPLL engine. Theory reasoning may be necessary to derive that the clause instance is unsatisfiable for the current partial assignment M. In particular, the literal instances do not always exist, and the definition of unit-propagating instances also allows new terms in $\ell_i\sigma$.

In the next section, we explain how to find conflicting and unit-propagating instances in the theory of equality $\mathcal{T}_E$. In Sect. 4.2, we then describe how the approach can be extended to the combination with linear arithmetic.

4.1 Finding Substitutions in the Theory of Equality

In the following we describe how to find conflicting and unit-propagating instances for quantified clauses in the theory of equality $\mathcal{T}_E$. As mentioned in the beginning of Sect. 4, the input formulas are preprocessed into conjunctive normal form. An uninterpreted predicate is treated as a function returning a Boolean and converted to an equality with the constant *true*. The preprocessor also applies destructive equality resolution (DER): clauses of the form $x \neq p \vee C$ where p does not contain x are replaced by the equivalent clause $C[x \mapsto p]$ where all occurrences of x are replaced by p. Also trivially false literals $x \neq x$ are removed from the clause. These formula simplifications are important for completeness as explained in Theorem 1 on page 14. From now on, let F be the preprocessed formula, and $C := C[x_1, \ldots, x_n]$ be a quantified clause in F with free variables $x_1, \ldots, x_n$.

Our approach to find conflicting and unit-propagating instances consists of three steps.

1. For each non-ground literal $p = p'$ or $p \neq p'$ in C, solve the E-matching problem for the multi-pattern p, p'. This finds ground substitutions σ for the variables in p, p' and congruent ground terms $t \sim p\sigma$, $t' \sim p'\sigma$.
2. Evaluate the equivalent literal $t = t'$ or $t \neq t'$ using information from the congruence closure theory solver.
3. Extract the common substitutions σ for all variables in C that are conflicting or unit-propagating.

We illustrate our approach with the help of the following example.

Example 2. We assume that the DPLL engine has already set the following literals to `true`.

$$M : f(a, b) = a, f(b, b) = b, f(b, c) = c, a = c$$

This literal assignment results in the E-graph displayed in Fig. 1. In the following we will show how to find conflicting and unit-propagating instances for the quantified clause

$$C : f(x, y) \neq c \vee f(y, z) \neq c \vee f(x, z) = c$$

with free variables x, y, z.

Step 1: Find Substitutions and Congruent Terms. The first step to find conflicting and unit-propagating instances for a quantified clause is to detect substitutions for which the value of the resulting instance in the current partial model can be determined without building the instance. This is the case if there exists a ground term $t \in T_F$ for each quantified term $p[x_1, \ldots, x_n] \in T_C$ such that t and $p[x_1, \ldots, x_n]$ are congruent under the substitution σ, i.e., $t \sim p[x_1, \ldots, x_n]\sigma$.

We search for such substitutions for each literal separately. In particular, for an equality or disequality literal in $\mathcal{T}_E$, i.e., a literal ℓ with underlying atom $p[x_1, \ldots, x_n] = p'[x_1, \ldots, x_n]$, we search for a substitution σ such that there exist terms $t \in T_F$ and $t' \in T_F$ with $t \sim p[x_1, \ldots, x_n]\sigma$ and $t' \sim p'[x_1, \ldots, x_n]\sigma$. These substitutions can be found by applying E-matching on the multi-pattern $p[x_1, \ldots, x_n], p'[x_1, \ldots, x_n]$. If one of the two patterns p, p' is ground, then we use only the other as a pattern. The substitutions found by E-matching are then stored in a table, which we will refer to as *substitution table*, where each row stands for a substitution σ and also stores the terms $t, t' \in T_F$ with $t \sim p[x_1, \ldots, x_n]\sigma$ and $t' \sim p'[x_1, \ldots, x_n]\sigma$. If a variable appearing in the clause C does not appear in the literal ℓ, the substitution for this variable is irrelevant for the literal, and the column corresponding to the variable is filled with asterisks.

E-matching can be implemented to work incrementally, and therefore, the substitution tables can be built up incrementally as well.

Example 3. Consider again the literal assignment M and the quantified clause C from Example 2. After E-matching with the patterns $f(x, y)$, $f(y, z)$, and $f(x, z)$, respectively, the substitution tables for the literals look as follows.

$f(x, y) \neq c$

x	y	z	congruent
a	b	$*$	$f(a, b), c$
b	b	$*$	$f(b, b), c$
b	c	$*$	$f(b, c), c$

$f(y, z) \neq c$

x	y	z	congruent
$*$	a	b	$f(a, b), c$
$*$	b	b	$f(b, b), c$
$*$	b	c	$f(b, c), c$

$f(x, z) = c$

x	y	z	congruent
a	$*$	b	$f(a, b), c$
b	$*$	b	$f(b, b), c$
b	$*$	c	$f(b, c), c$

Step 2: Evaluate Literal Instances. For each literal, we can now determine which value the literal instance resulting from a substitution would have in the current partial assignment. The value of a literal under a given substitution can be determined by checking equality or disequality for the congruent terms $t, t' \in T_F$ with $t \sim p\sigma$ and $t' \sim p'\sigma$. As mentioned in Sect. 3.2, the partial assignment for the theory of equality is represented by the E-graph, together with a set of disequality literals that are currently set to true by the DPLL engine.

If the literal is implied by the current partial assignment, the corresponding instance is irrelevant for the current state as the clause is already satisfied. If the negation of the literal is implied, it can lead to a conflicting or unit-propagating instance. If neither the literal nor its negation is implied, it is a possible candidate for unit-propagation. For an equality literal $\ell : p[x_1, \ldots, x_n] = p'[x_1, \ldots, x_n]$, the value $val_\sigma(\ell)$ under a substitution σ with congruent terms $t \sim p[x_1, \ldots, x_n]\sigma$ and $t' \sim p'[x_1, \ldots, x_n]\sigma$ is defined as follows.

$$val_\sigma(\ell) = \begin{cases} \texttt{irrel} & \text{if the congruence classes } [t]_= \text{ and } [t']_= \text{ are equal} \\ \texttt{false} & \text{if a disequality } t_1 \neq t_2 \text{ between terms } t_1 \in [t]_= \\ & \text{and } t_2 \in [t']_= \text{ is set} \\ \texttt{unit} & \text{otherwise} \end{cases}$$

For a disequality literal $\ell : p \neq p'$, the value $val_\sigma(\ell)$ is defined analogously with first and second case swapped.

We evaluate each row of the substitution table and get a *literal value table* where each row represents a substitution and the corresponding literal value. As in the substitution tables, a column full of asterisk indicates a variable that does not appear in the literal. We add a row full of asterisks in the end of the table that represents all substitutions where E-matching has not found congruent terms.

If all other literals evaluate to false under such a substitution, this substitution leads to a unit-propagating instance that propagates a literal on new terms. While it may sometimes be helpful to propagate these literals, it often leads to many unnecessary propagations and can even lead to matching loops. Therefore, we have an option to either mark this row as unit or as irrelevant depending on whether equalities on unknown terms should be propagated.

Example 4. The tables T_1, T_2, and T_3 below are the literal value tables corresponding to the substitution tables of $f(x, y) \neq c$, $f(y, z) \neq c$, and $f(x, z) = c$, respectively, from Example 3.

T_1

x	y	z	value
a	b	$*$	`false`
b	b	$*$	`unit`
b	c	$*$	`false`
$*$	$*$	$*$	`unit/irrel`

T_2

x	y	z	value
$*$	a	b	`false`
$*$	b	b	`unit`
$*$	b	c	`false`
$*$	$*$	$*$	`unit/irrel`

T_3

x	y	z	value
a	$*$	b	`irrel`
b	$*$	b	`unit`
b	$*$	c	`irrel`
$*$	$*$	$*$	`unit/irrel`

Step 3: Evaluate Clause Instances and Extract Substitutions. Once a table for each literal has been built, these tables are combined in order to determine the value of the clause instances under the substitutions found for the literals. As we are only interested in conflicting and unit-propagating instances, we consider substitutions where a literal instance has value `irrel`, or where two or more literals have value `unit`, to be irrelevant. Thus, we distinguish three values for the clause tables: `false` if all literals evaluate to `false` under a substitution, `unit` if all but one literal evaluate to `false` under a substitution and the remaining literal evaluates to `unit`, and `irrel` for all other cases.

The clause value tables are computed as follows. A new clause table starts with a row full of asterisks mapping to the value `false`, i.e., it looks as follows.

$x_1 \dots x_n$	value
$* \dots *$	`false`

Then for each row in the clause table and the next literal table, we check if they are compatible and combine them. Compatible means, for each variable the terms are currently congruent, or for one table the variable does not occur in the substitution, i.e., there is an asterisk in the corresponding column. If the rows are compatible, the substituted terms are combined by keeping the terms from the first table, except for the positions marked with an asterisk, where we use the terms from the second row. The values of the tables are combined according to the mapping

$$\begin{aligned}(x, \texttt{false}) &\mapsto x\\ (\texttt{false}, x) &\mapsto x\\ \text{else} &\mapsto \texttt{irrel}\end{aligned}$$

A row becoming irrelevant can be dropped. The clause table is combined with the literal table for each literal in the clause.

The result is a table that contains a row for each conflicting or unit-propagating instance with values `false` or `unit`, respectively, and a row full of asterisks with value `irrel`.

Example 5. We now combine the literal value tables from Example 4 step by step, i.e., in the first step we combine the default clause table T_0 with the table T_1 for $f(x, y) \neq c$, then we combine the result with the table T_2 for $f(y, z) \neq c$ and finally with T_3 for $f(x, z) = c$.

$T_0 + T_1$

x	y	z	value
a	b	$*$	`false`
b	b	$*$	`unit`
b	c	$*$	`false`
$*$	$*$	$*$	`unit/irrel`

$T_0 + T_1 + T_2$

x	y	z	value
a	b	b	`unit`
a	b	c	`false`
a	b	$*$	`unit/irrel`
b	b	c	`unit`
b	c	b	`false`
b	c	$*$	`unit/irrel`
$*$	a	b	`unit/irrel`
$*$	b	c	`unit/irrel`
$*$	$*$	$*$	`irrel`

$T_0 + T_1 + T_2 + T_3$

x	y	z	value
a	b	c	`unit/irrel`
b	c	b	`unit`
$*$	$*$	$*$	`irrel`

Note that the row for the substitution $\{x \mapsto b, y \mapsto c, z \mapsto b\}$ in $T_0 + T_1 + T_2$ and $T_0 + T_1 + T_2 + T_3$ results from combining $\{x \mapsto b, y \mapsto c, z \mapsto *\}$ from $T_0 + T_1$ with $\{x \mapsto *, y \mapsto a, z \mapsto b\}$ from T_2, which are compatible because $a \sim c$ holds.

The substitution $\sigma = \{x \mapsto b, y \mapsto c, z \mapsto b\}$ produces a unit-propagating instance for C containing the term $f(c, b)$ which is a new term, but congruent to the term $f(a, b)$ found for the literal $f(y, z) \neq c$. The substitution $\sigma = \{x \mapsto a, y \mapsto b, z \mapsto c\}$ also produces a unit-propagating instance for C, but this instance contains the new term $f(a, c)$ that is not congruent to any known term so far.

If we consider $M' : M, a \neq b$, the table T_3' for the literal $f(x, z) = c$ changes and so does the final clause table.

T_3'

x	y	z	value
a	$*$	b	`irrel`
b	$*$	b	`false`
b	$*$	c	`irrel`
$*$	$*$	$*$	`unit/irrel`

$T_0 + T_1 + T_2 + T_3'$

x	y	z	value
a	b	c	`unit/irrel`
b	c	b	`false`
$*$	$*$	$*$	`irrel`

The instance $C\sigma$ with $\sigma = \{x \mapsto b, y \mapsto c, z \mapsto b\}$ is conflicting for M'.

Instantiation. After computing the clause value tables, the conflicting instances are built. These instances often create new terms and literals, because with E-matching the term $p\sigma$ may not exist in the E-graph. However, in this case the solver for congruence closure can propagate these new literals to `false`.

The quantifier theory waits until all literals are propagated by the theory of equality and only then returns the instance as conflict clause.

If no conflicting instances are found, the unit-propagating instances are built. Again these may contain new terms and literals that the solver for congruence closure will propagate to `false`. Only when the instances become unit clauses, they will be returned by the quantifier theory solver.

Theoretical Results. The presented method is incomplete in the sense that it cannot detect all conflicting or unit-propagating instances for the theory of equality $\mathcal{T}_E$. There are two reasons for the incompleteness:

1. The congruence closure algorithm does not propagate all implied disequalities. For example if $f(a) \neq f(b)$ is set, the disequality $a \neq b$ cannot be detected in the E-graph. Thus, when evaluating the literal $a = b$ it may incorrectly be classified as `unit` instead of `false`.
2. Some congruences cannot be found by E-matching because no congruent terms exist in the E-graph. In particular, the equality $(f(p_1, \ldots, p_n) = f(p'_1, \ldots, p'_n))\sigma$ holds if $p_i\sigma$ and $p'_i\sigma$ are congruent for $1 \leq i \leq n$, but there need not be congruent terms in the E-graph for $f(p_1, \ldots, p_n)\sigma$ and $f(p'_1, \ldots, p'_n)\sigma$.

The first reason can be avoided. Instead of just asking for the existence of a disequality edge between two terms, one could check if adding an equality between the terms and propagating all congruences leads to a conflict. However, this contradicts our main goal, which is to make the solver fast enough that it can find conflicting instances eagerly.

To understand the second reason, we investigate the cases in which an instance of a literal is conflicting, i.e., where $M \models_{\mathcal{T}_E} \neg\ell\sigma$ holds. We distinguish four cases. In the first case, E-matching is sufficient to find the conflicting instance. The second to fourth cases can be avoided by preprocessing as explained later.

Lemma 1. *Let M be a consistent (partial) literal assignment, and let ℓ be a literal. Assume that all ground terms occurring in the literals in M and in the literal ℓ are present in the E-graph.*

If $M \models_{\mathcal{T}_E} \neg\ell\sigma$ holds for a substitution σ, then

1. *the literal ℓ is an equality $p = p'$ or a disequality $p \neq p'$, there are terms t, t' in the E-graph with $t \sim p\sigma$ and $t' \sim p'\sigma$, and the corresponding disequality or equality between t and t' is implied by M, or*
2. *the literal ℓ is a disequality $x \neq p$, where p does not contain x, or*
3. *the literal ℓ is a disequality $x \neq x$, or*
4. *the literal ℓ is a disequality $f(p_1, \ldots, p_n) \neq f(p'_1, \ldots, p'_n)$ and for all corresponding subterms, $M \models_{\mathcal{T}_E} (p_i = p'_i)\sigma$ holds.*

Proof. $M \models_{\mathcal{T}_E} \neg\ell\sigma$ holds, iff $M \cup \{\ell\sigma\} \models_{\mathcal{T}_E} \bot$. It is well-known that the congruence closure algorithm can find all ground conflicts. First, consider the case that $\ell : p = p'$ is an equality literal. The congruence closure algorithm would create new terms $p\sigma$, $p'\sigma$ and add an equality edge between them. Assume that case 1 does not apply. This means, $p\sigma$ or $p'\sigma$ are not congruent to an existing term; we assume w.l.o.g. that this holds for $p\sigma$. So before the equality literal $p\sigma = p'\sigma$ is considered, the node $p\sigma$ would not be equivalent to any other term in the E-graph and there would not be any function application on $p\sigma$. Hence, the step in the congruence algorithm that merges $p\sigma$ and $p'\sigma$ would not introduce any more congruences and it would only merge the fresh node $p\sigma$. Thus, any conflict found by the congruence closure algorithm for $M \cup \{\ell\sigma\}$ would already be present in M. We assumed that M is consistent, so this is a contradiction.

Let now $\ell : p \neq p'$ be a disequality literal. We assume that M is consistent, but $M \cup \{\ell\sigma\}$ is not. Hence, adding $p\sigma$ and $p'\sigma$ to the E-graph would derive an equality between these terms. If there is an equality between the new terms and some already existing terms, then we are in case 1. Otherwise, the equality can only follow by congruence, or $p\sigma$ and $p\sigma'$ are identical. If they are identical constants, the literal ℓ must be of form $x \neq y$ or $x \neq x$, as we assume that all ground terms are present in the E-graph. Hence, we are in case 2 or 3. Otherwise, both $p\sigma$ and $p'\sigma$ are function applications and their arguments are equal. Assume that we are not in case 4. Then either p or p' must be a variable, w.l.o.g. assume $p = x$. If we are not in case 2 or 3, then p' is a function application on a term that contains x. But then $\sigma(x) = p'(\sigma(x))$ must follow from M. This can only be the case if M contains an equality for a term congruent to $\sigma(x)$ or one of its parents. But then $\sigma(x)$ must have a congruent term t in the E-graph, so we were in case 1 all along. □

This lemma shows that the following preprocessing is sufficient to find all conflicting instances. Let $\mathcal{C}$ be a set of quantified clauses. We create a new set of preprocessed clauses *preprocess*($\mathcal{C}$) by exhaustively applying the following rules on $\mathcal{C}$:

1. If there is a clause $C \in \mathcal{C}$ of the form $C : x \neq x \vee C'$, remove it and add the clause C' instead.
2. If there is a clause $C \in \mathcal{C}$ of the form $C : x \neq p \vee C'$ where p does not contain x, remove it and add $C'[x \mapsto p]$ instead (DER).
3. If there is a clause $C \in \mathcal{C}$ of the form $C : f(p_1, \dots, p_n) \neq f(p'_1, \dots, p'_n) \vee C'$, *copy* it and add the clause $p_1 \neq p'_1 \vee \dots \vee p_n \neq p'_n \vee C'$.

Note that the third rule is sound because $f(p_1, \dots, p_n) \neq f(p'_1, \dots, p'_n)$ implies the disjunction $\bigvee p_i \neq p'_i$. The preprocessor must still keep the original clause, in case the literal is false due to an explicit disequality that can be found by E-matching.

After preprocessing, every conflict on a single clause instance can be found with E-matching:

Theorem 1. *Let M be a consistent (partial) literal assignment and $\mathcal{C}$ a set of quantified clauses. Let there be a clause $C \in \mathcal{C}$ and a substitution σ with $M \models_{\mathcal{T}_E} \neg C\sigma$. Then there is a clause $C' \in$ preprocess($\mathcal{C}$), such that for each literal in C' of the form $p = p'$ (resp. $p \neq p'$) there are E-matching equivalent terms t, t' with $t \sim p\sigma$ and $t' \sim p'\sigma$ and $M \models_{\mathcal{T}_E} \neg(t = t')$ (resp. $M \models_{\mathcal{T}_E} \neg(t \neq t')$).*

4.2 Extension to Linear Arithmetic

The approach described in the previous section can be extended to formulas in other theories. In this section we consider the extension of our approach to linear arithmetic. Finding congruent terms in general is difficult and costly, in particular for literals containing terms that mix arithmetic and functions like, e. g., $f(g(x)+h(y))$. In the following, we describe some extensions that we think are useful and can still be treated with reasonable cost.

The first extension is to treat literals that contain arithmetic only at top level, i.e., literals of the form $c_0+\sum c_i p_i[x_1,\ldots,x_n]=0$ or $c_0+\sum c_i p_i[x_1,\ldots,x_n]\leq 0$, where the p_i are terms of $\mathcal{T}_E$. In Step 1, we take the multi-pattern $p_1, p_2, \ldots$ to find congruent terms with E-matching. In Step 2, the value of the literal under a substitution σ with congruent terms $t_i \sim p_i[x_1,\ldots,x_n]$ can then be determined as follows. For $\ell : c_0+\sum c_i p_i[x_1,\ldots,x_n]\leq 0$, we check if there exist any bounds on the term $c_0+\sum c_i t_i$. If the term has an upper bound $u \leq 0$, then $val_\sigma(\ell) = \texttt{true}$, if it has a lower bound $l > 0$, then $val_\sigma(\ell) = \texttt{false}$, and $val_\sigma(\ell) = \texttt{unit}$ otherwise. For $\ell : c_0+\sum c_i p_i[x_1,\ldots,x_n]=0$, if $c_0+\sum c_i t_i$ has an upper bound u and a lower bound l, and $u=l=0$, then $val_\sigma(\ell) = \texttt{true}$. If it has a lower bound $l > 0$ or an upper bound $u < 0$, then $val_\sigma(\ell) = \texttt{false}$, and $val_\sigma(\ell) = \texttt{unit}$ otherwise. Our solver is based on the Simplex algorithm described in [7]. It uses the bound refinement method described there to propagate bounds that are implied by the current state of the tableau. While this is inherently incomplete, it is fast and the bounds are refined incrementally.

Another important extension is to treat arithmetical literals such as $x < t$. These literals occur frequently when reasoning about arrays, and fall into the decidable array property fragment [2]. In principle, any substitution for x can be evaluated using upper and lower bounds as above, but we restrict the substitutions to consider as follows. For a clause C containing arithmetical literals and other literals of the types above, we first build the partial clause value table by evaluating the other literals as described before. Then for each variable x in C, we collect a set R_x of relevant terms as follows:

1. $R_x := R_x \cup \{t \mid \sigma(x) = t$ for σ with partial clause value $\texttt{false}$ or $\texttt{unit}\}$.
2. If the clause contains a literal $x < t$ or $t < x$, then $R_x := R_x \cup \{t\}$.
3. If the clause contains a literal $x = t$, then $R_x := R_x \cup \{t+1, t-1\}$.
4. If the clause contains $x < y$ or $y < x$, then $R_x := R_x \cup R_y$.

This is inspired by [8]. Given those sets, the substitutions we consider for an arithmetical literal ℓ of form $x < t$, $t < x$ or $x = t$ are $\{\sigma_\ell = \{x \mapsto t\} \mid t \in R_x\}$, and for $\ell : x < y$ they are $\{\sigma_\ell = \{x \mapsto t, y \mapsto t'\} \mid t \in R_x, t' \in R_y\}$.

5 Implementation and Experiments

We implemented the presented method in the SMT solver SMTInterpol.[2] SMTInterpol is a DPLL($\mathcal{T}$)/CDCL based solver that supports the ground

[2] https://ultimate.informatik.uni-freiburg.de/smtinterpol/

fragments of the theory of equality, the theory of linear integer arithmetic, linear rational arithmetic, mixed linear integer-rational arithmetic, the theory of arrays with extensionality and constant arrays, and their combinations.

We implemented the quantifier support as a theory solver in the DPLL($\mathcal{T}$) framework. The DPLL engine informs all theory solvers about the literals that are currently set to true. Before each decision, all theory solvers search for conflicts and unit clauses in a *checkpoint*. If a theory solvers returns a new unit clause, it can be used to propagate new literals and thus avoid wrong decisions. Similarly, a conflict clause allows to backtrack immediately without doing further decisions. When the DPLL engine has assigned a truth value to all literals, a *final check* is performed where the theory solvers should check their model.

The solver for quantified formulas keeps a list of all quantified clauses and creates instances of them on the fly. The quantifier solver has two different settings to determine when to create new clauses. In the eager setting, it creates new clauses in the checkpoint before each decision of the DPLL engine, in the lazy setting, it only creates new clauses in the final check when all existing ground literals were decided by the DPLL engine. When our method finds a conflicting or unit-propagating instance, this instance is built. If the instance is a conflict in the sense that all literals are already set to false, it is returned immediately. If the instantiation creates new terms and literals, it will cause other theory solvers, in particular the congruence closure solver, to propagate congruences and truth values for the new literals. As soon as all but one literal in the instance are propagated to false, our quantifier solver can give the instantiated clause to the DPLL engine as a unit clause.

If our E-matching based procedure does not find any conflict or unit clauses in the final check, our new quantifier solver has to do more extensive checks to determine if the formula is satisfied. It checks the instances created from the substitution set described in [8] for formulas in the almost uninterpreted fragment. To ensure completeness it tries substitutions on "older" terms first in order to enumerate the terms in a systematic way, similarly to [13]. In particular this means that substitution with terms that occur in the input formula are preferred over terms that are created by the quantifier solver itself during the solving process. For terms with the same age, the final check prefers instances that are unit-propagating (and that were not found earlier, because they create new terms that are not equivalent to existing terms). If no such instances are found, any instance that is not yet satisfied is created, preferring substitutions that do not create new terms. This allows the solver to return "satisfiable" for formulas within the almost uninterpreted fragment when it has checked that all instances resulting from these substitution are satisfied. In case a problem contains literals outside the almost uninterpreted fragment, the solver will never return "satisfiable", but "unknown" if it cannot derive a conflict.

We implemented E-matching to find substitutions as described in Step 1 in Sect. 4 to work in an incremental way, similar to [3]. For each quantified clause, we choose as multi-pattern the set of all sub-terms occurring in the clause, i.e., the instantiation only creates a new term if there is an equivalent existing term.

It uses triggers within the solver for the theory of equality, that report new terms for sub-patterns as they are merged into the relevant congruence class, and cause the matching process to continue only then. As soon as the multi-pattern is matched, the substitution is saved. Any substitution found by E-matching is kept until a conflict is detected and the DPLL engine backtracks to a point where the pattern no longer matches.

To efficiently implement the substitution tables, the literal value tables and the clause value tables, we use directed acyclic word graphs (DAWGs). These are useful to quickly combine a clause value table with the next literal value table, especially in the presence of columns with asterisks.

In order to evaluate the usefulness of our algorithm, we compare our implementation against E-matching based instantiation. We tested four different settings: The settings "conflict/unit-eager" and "conflict/unit-lazy" use the presented algorithm to search for conflicting and unit-propagating instances. Both settings do not create new terms (up to congruence) and always prefer conflicting instances over unit-propagating instances. As the names suggest, the setting "conflict/unit-eager" runs our algorithm as described above in the checkpoint. The setting "conflict/unit-lazy" runs our algorithm in the final check, i.e., after a complete ground model has been built. The settings "E-matching-eager" and "E-matching-lazy" use our implementation of the E-matching algorithm in a more traditional way. They use the same multi-pattern as our presented algorithm, and build all instances where the multi-pattern was matched. This means that no new terms (up to congruence) are built in these settings. As above, the setting "E-matching-eager" searches for instances in the checkpoint while "E-matching-lazy" searches for instances in the final check.

We did two experiments to evaluate these algorithms [9]. First, we ran them on all SMT-LIB benchmarks in the logic UF on an AMD Ryzen Threadripper 3970X 32-Core CPU with 3.7 GHz, using 8 cores in total, and 15 GB RAM given to the solver. We set the timeout to 24 s. Second, we ran the UF division with the settings used for the SMT-COMP 2020[3] on the StarExec cluster[4] [16], including the same benchmark selection and the same scrambler with the same seed. The only difference was that we reduced the timeout to 10 min (instead of 20 min). The SMT-COMP benchmarks omit all benchmarks that were solved by all solvers in less than one second in the previous years and randomly selected 40 % of the remaining benchmarks. We also ran the solvers cvc4 version 1.8 and z3 version 4.8.8 on the SMT-COMP benchmarks with the default settings. The results are summarized in Tables 1 and 2.

Table 1 shows that the settings that produce only conflict/unit instances solve more benchmarks than the settings that produce all E-matching instances. The difference is even more pronounced on the SMT-COMP benchmark set where easy benchmarks were removed. The difference between eager and lazy settings is only small, but in our experiments eager was slightly better. This shows that the additional overhead from doing conflict search before each decision is more than

[3] https://smt-comp.github.io/2020/
[4] https://www.starexec.org

compensated by the reduced search space. The evaluation also shows that our solver is not yet competitive with CVC4 and Z3. The simple E-matching strategy that requires all subterms to exists and the simple enumeration of terms by age as fallback strategy is no match to the more fine-tuned and diverse strategies in CVC4 and Z3.

Table 1. Number of benchmarks solved for each solver setting, with 24 s timeout (on all UF benchmarks) and with 10 min timeout (on the SMT-COMP 2020 UF selection). The settings "c/u" use our presented method to produce only conflict and unit-propagating instances. The setting "eager" produces all E-matching instances before every decision, "lazy" only when all literals were decided.

Solver	Setting	UF(all) 24 s timeout	UF(SMT-COMP) 10 min timeout
SMTInterpol	c/u-eager	2120/7668	211/2291
SMTInterpol	c/u-lazy	2105/7668	207/2291
SMTInterpol	eager	2011/7668	165/2291
SMTInterpol	lazy	1998/7668	161/2291
CVC4	–		514/2291
Z3	–		408/2291

In Table 2 we compare the number of instances produced by SMTInterpol in the different settings and by CVC4 and Z3. The numbers were obtained by dumping the statistics after the run. To make the numbers comparable, we only consider those benchmarks from the SMT-COMP benchmark set where all solvers in all settings could prove unsatisfiability. For SMTInterpol we also count the number of instances that were used in the final proof of unsatisfiability. The first apparent result is that only a fraction of the instances were needed. This shows the importance of choosing the right instances. The settings that produce only conflict/unit-propagating instances save a lot of instances that were not needed in the proof of unsatisfiability. Interestingly, CVC4 creates even fewer instances. Note that CVC4 also uses conflict based instantiation techniques. One reason that it needs even fewer instances than our approach might be that our enumeration strategy in the final check needs longer to find the right instances. Another reason is that CVC4 does not split large quantified formulas into several clauses (and thus needs only one instance where we may need one instance for each produced clause). The solver Z3, which does not use conflict based instantiation techniques, produces many more instances. The average is exaggerated due to one benchmark where it produces more than 5.7 million instances, but the median is also higher than in our conflict/unit-propagating settings.

For SMTInterpol we distinguish instances created by E-matching or by conflict/unit-propagation from instances created by the final enumeration step. This is depicted in the table as 588(230+358) denoting that 230 instances were

Table 2. Average and median number of instances created by the solvers on SMT-COMP 2020 benchmarks. For SMTInterpol also the average number of instances used in the proof of unsatisfiability is given. This statistic was generated for the 86 benchmarks that every solver could solve.

Solver	Setting	Avg. created instances	Median created instances	Avg. used instances
SMTInterpol	c/u-eager	$588(230 + 358)$	56	$8(3 + 5)$
SMTInterpol	c/u-lazy	$545(195 + 351)$	56	$8(3 + 5)$
SMTInterpol	eager	$1455(1121 + 333)$	195	$8(3 + 5)$
SMTInterpol	lazy	$1450(1123 + 327)$	217	$8(3 + 5)$
cvc4	–	216	14	?
z3	–	83186	129	?

created by conflict/unit-propagation and 358 by the final enumeration. The results show that several necessary instances can only be found by enumeration, because they need to create new terms that are not equivalent to existing terms.

6 Related Work

Closest related to the presented approach is the work of Reynolds et al. [15]. The authors present a method to find conflicting substitutions for the theory of equality and show the effectiveness of their approach. For a quantified formula, they construct a set of equalities and disequalities such that a solution for the constraints in this set yields a conflicting substitution. While our method works only on clauses, their method works on general quantified formulas. The classification is done implicitly while constructing those constraint sets, but avoids introducing auxiliary functions. Some of the constructed equality constraints represent the problem of matching subterms of more complicated non-ground terms, but the algorithm does not use E-matching. The types of literals that pose a problem to our approach (i.e., congruences without congruent existing terms) cannot be detected by their method either. The authors also describe how the method can be used to find so-called constraint-inducing substitutions that produce instances that are not conflicting, but that can derive new information about existing terms. This is similar to our search for unit-propagating instances, but does not allow to find propagations on new terms. The main difference is that our approach is incremental and can therefore detect conflicting and unit-propagating instances early in the solving process.

Congruence closure with free variable (CCFV) [1] is a calculus for solving the so-called E-ground (dis)unification problem. Given a ground model, it tries to build a substitution for a quantified formula such that the ground model satisfies the corresponding instance, by decomposing the goal into smaller constraints. It can also be used to search for a conflicting substitution and can find all conflicting substitutions for the theory of equality, if the congruence closure propagates all disequalities. The method is not incremental, i.e., it needs to rerun completely if an equality or disequality literal is added.

A completely different approach to derive new facts from quantified clauses is the DPLL(Γ) calculus [4]. This approach combines the superposition calculus tightly with the DPLL($\mathcal{T}$) framework. Literals decided by the DPLL engine are used in the superposition solver to derive new quantified clauses. When the superposition solver finds a conflict, it can build by collecting all ground equalities used to derive the conflict. DPLL(Γ) can also propagate new ground literals using quantified clauses. While the approach is much more powerful and can even detect conflicts involving several clauses, this comes at the price of memory overhead. The superposition solver can propagate an arbitrary number of derived clauses when searching for conflicts. This is in contrast to the DPLL($\mathcal{T}$) framework where only a detected conflict triggers learning a new clause.

7 Conclusion and Future Work

We presented a new approach to find conflicting and unit-propagating instances of quantified formulas. The basic idea is to split this search in a part that searches for ground terms that are congruent to the quantified terms in a clause, and then evaluate the instances with the use of these terms before creating them. For the first part, we use E-matching, which can be implemented in an incremental way and avoids duplicating work when the E-graph changes. The evaluation can be done per literal such that the method can also detect instances that propagate literals on both known and new terms. The presented method has been implemented in the SMT solver SMTInterpol. We showed that by only producing conflicting and unit-propagating instances we can solve more benchmarks than by producing all instances found by E-matching. We also showed that the overhead to find these conflicting instances is small enough to run it in an eager setting before every decision. Therefore, we can tightly integrate quantifier reasoning in the DPLL($\mathcal{T}$) framework. We believe that the method can easily be implemented into other solvers using E-matching based instantiation, since E-matching can already report the equivalent terms needed for evaluating instances.

We also presented some extensions to the theory of linear arithmetic, and plan to extend the method further. For instance, with a solver for the theory of equality that supports *offset equalities* [6], literals such as $f(x) = g(x + 1)$ can easily be evaluated as described in Sect. 4.

The method is incomplete and must be complemented with a method that checks the model once all literals are assigned a truth value by the DPLL engine. This complementary method can have a strong influence on our presented method, in particular, if it creates many new terms. We plan to implement a version of model-based quantifier instantiation [8] in the future.

We also plan to implement a version of our method that does not create new literals for conflicting instances at all. Instead of creating the conflicting instance, it creates the clause with the existing equivalent literals enriched by the equality literals from the E-graph that were needed to prove the equivalence. Currently the quantifier solver has to wait for the congruence closure to prove that the conflicting instance is a conflict clause. We also expect that this approach keeps the E-graph small by not creating many congruent terms.

References

1. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017, Part II. LNCS, vol. 10206, pp. 214–230. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
2. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005). https://doi.org/10.1007/11609773_28
3. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
4. de Moura, L., Bjørner, N.: Engineering DPLL(T) + Saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_40
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
6. Dutertre, B., de Moura, L.: The Yices SMT solver. Technical report, SRI International (2006). https://yices.csl.sri.com/papers/tool-paper.pdf
7. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_11
8. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
9. Hoenicke, J., Schindler, T.: Artifacts for incremental search for conflict and unit instances of quantified formulas with E-matching. Technical report, Zenodo (2021). https://doi.org/10.5281/zenodo.4277777
10. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM **27**(2), 356–364 (1980)
11. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM **53**(6), 937–977 (2006)
12. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 335–367. Elsevier and MIT Press, New York (2001)
13. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018, Part II. LNCS, vol. 10806, pp. 112–131. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_7
14. Reynolds, A., King, T., Kuncak, V.: Solving quantified linear arithmetic by counterexample-guided instantiation. Form. Methods Syst. Des. **51**(3), 500–532 (2017). https://doi.org/10.1007/s10703-017-0290-y

15. Reynolds, A., Tinelli, C., de Moura, L.M.: Finding conflicting instances of quantified formulas in SMT. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 195–202. IEEE (2014)
16. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28