Quantifier Simplification by Unification in SMT

Pascal Fontaine[1,2] and Hans-Jörg Schurr[1]

[1] University of Lorraine, CNRS, Inria, and LORIA, Nancy, France
hans-jorg.schurr@inria.fr
[2] Université de Liège, Belgium
pascal.fontaine@uliege.be

Abstract. Quantifier reasoning in SMT solvers relies on instantiation: ground instances are generated heuristically from the quantified formulas until a contradiction is reached at the ground level. Current instantiation heuristics, however, often fail in presence of nested quantifiers. To address this issue we introduce a unification-based method that augments the problem with shallow quantified formulas obtained from assertions with nested quantifiers. These new formulas help unlocking the regular instantiation techniques, but parsimony is necessary since they might also be misguiding. To mitigate this, we identify some effective restricting conditions. The method is implemented in the veriT solver, and tested on benchmarks from the SMT-LIB. It allows the solver to prove more formulas, faster.

Keywords: SMT · quantifier instantiation · theorem proving

1 Introduction

Satisfiability modulo theories (SMT) solvers are successfully used as back-ends for formal method applications, within interactive proof assistants or verification platforms. SMT solvers based on the CDCL($\mathcal{T}$) calculus [5] excel at handling quantifier-free problems with theories — SMT problems with thousands of assertions are frequent. While originally SMT solvers were mostly applied on such problems, an increasing number of applications require some support for quantifier reasoning. Their main approach to handle quantifiers is *quantifier instantiation*. This approach separates the quantified assertions from the ground part of the problem. Whenever the solver finds a model for the ground part, it generates new ground instances of the quantified formulas. This is repeated until the ground solver determines that the ground problem is unsatisfiable. When done fairly, this approach is refutationally complete for many theories and due to the strength of ground solving it is also very powerful in practice. SMT solvers use multiple instantiation strategies to find these instances. The main challenge is to find the right instances without misguiding or overwhelming the solver. Often one can observe some kind of *butterfly effect*: if the instantiation methods are unlucky, the solver might be misguided to explore a large set of irrelevant instances and reach the solving timeout. In this regard, every strategy has its own strength and weaknesses (Section 2).

If the problem contains a quantified lemma that also occurs, for example, as an antecedent for another formula, the common instantiation methods often fail to quickly produce the right instances. This structure is quite typical of problems generated by interactive theorem provers — a domain to which the SMT solver veriT has been successfully applied [9, 19]. The following toy example illustrates this issue. We will use it to illustrate various ideas.

Example 1.

$$\forall x.\, P(x) \to P(f(x, c)) \tag{1}$$

$$\neg P(c) \lor (\forall y.\, (\forall z.\, P(z) \to P(f(z, y))) \to \neg P(y)) \tag{2}$$

$$P(c) \tag{3}$$

This problem is trivially unsatisfiable: when y is set to c, assertion 1 occurs as the antecedent of the implication in assertion 2, so $\neg P(c)$ is a direct consequence of the first two assertions, in contradiction with the third. As described in Section 2.3, all major instantiation techniques fail to directly produce the correct instances for this problem. Because SMT solvers typically only perform very limited preprocessing on quantified formulas and especially do not calculate a full clause normal form, the instantiation methods fail to recognize and exploit the fact that assertion 1 and the antecedent in assertion 2 are so similar. Since the instantiation methods do not produce the correct instances early, the SMT solver will need multiple instantiation rounds to solve the problem. This can lead to the butterfly effect mentioned above. Real world examples are usually more complex. For example, there are often many ground terms which mislead the instantiation heuristics. Furthermore, the assertions in this example are Horn clauses and could be handled by specialized reasoning. Practical problems, however, are not restricted to Horn clauses.

In CDCL($\mathcal{T}$) quantified formulas are considered black boxes, and are abstracted as propositional variables in the propositional abstraction of the input formula. These propositional literals are generally of no value to the ground solver. We here make use of them to simplify larger formulas. To solve the example above we identify the occurrence of the unit assertion 1 within assertion 2. By using unification we can eliminate this quantified subformula. The result after simplification is the ground formula $\neg P(c)$. After this formula is conjoined to the problem, it is trivially contradictory. In the general case, we use asserted quantified formulas to soundly simplify nested formulas and augment the problem with the result. We propose multiple variants of the core procedure (Section 3).

So far, techniques inspired by resolution-based theorem provers are underrepresented in SMT solvers. Systems such as DPLL(Γ) [14], DPLL($\Gamma + \mathcal{T}$) [6], and AVATAR [21] combine the inference system of theorem provers with the CDCL(T) transition system on a fundamental level, but the combination is coarse — in those systems the two worlds work side-by-side in tandem. Instead, our unification-based method is a lightweight and easily implemented prepro-

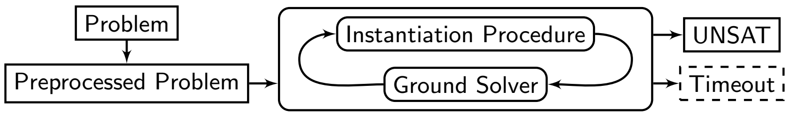

Fig. 1. The instantiation loop of an SMT solver refuting a problem.

cessing technique that solves some concrete shortcoming of current instantiation techniques.

We implemented our method in the SMT solver veriT [7]. To ensure the process is fast, we use a standard term index and unification algorithm which we extended to handle the presence of strongly quantified variables (Section 4).

The evaluation shows that our technique enables veriT to solve benchmarks not solved by any strategy before. When applicable, the method often allows veriT to solve problems within a short timeout. The different variants of the simplification process are useful within a strategy schedule (Section 5).

2 CDCL($\mathcal{T}$) and Quantifier Instantiation

Figure 1 shows the operation of a typical SMT solver when refuting a problem. It first preprocesses the input problem (Section 2.2). Then two procedures together refute the problem: the ground solver either refutes the problem on the ground level, or finds a ground model. If a model is found the instantiation procedure creates new ground lemmas (Section 2.3).

2.1 Preliminaries

We use the many-sorted first-order logic with equality as defined in the SMT-LIB standard [4] and assume the reader is familiar with the notions of signature, term, free and bound variable, quantified and ground formula, literal, and substitution. We use x, y, z to denote variables; s, t to denote terms; φ, ψ to denote formulas (i.e., terms of sort Bool); P to a predicate (i.e., a function with codomain sort Bool); and c to denote constants. To denote the substitution which replaces a variable x with a term t we write $[t/x]$. As usual, σ stands for a substitution. We write $\bar{t}$ for the sequence of terms $t_1, \ldots, t_n$ for an unspecified $n \in \mathbb{N}^+$ that is either irrelevant or clear from the context. Hence, $\forall \bar{x}.\varphi$ corresponds to a term $\forall x_1, \ldots, x_n.\, \varphi$. We write $\mathbf{free}(t)$ to denote the free variables of a term t. The set $\mathcal{T}(S)$ is the set of all subterms of the terms in S. We omit sorts when they are clear from the context and assume that sort constraints are always respected, e.g., substitutions only use terms of the same sort as the substituted variable.

Like in the SMT-LIB standard, the signature Σ always contains a sort Bool, two constants $\top$ and $\bot$, the usual Boolean connectives, and a family of predicate symbols ($\approx : \tau \times \tau \rightarrow \mathsf{Bool}$) interpreted as equality for each sort τ. A trimmed formula is the generalization of the notion of atom to arbitrary formulas: $\mathbf{trim}(\varphi)$

is the formula φ after removing all leading negations. For example, $\mathbf{trim}(\neg\neg(\varphi_1 \vee \neg\varphi_2))$ is $\varphi_1 \vee \neg\varphi_2$. The *polarity* $\mathbf{pol}(\varphi) \in \{+, -\}$ of a formula φ is $-$ (*negative*) if $\mathbf{trim}(\varphi)$ removes an odd number of negations and $+$ (*positive*) otherwise.

We write $t[\,]$ for a term with a hole and $t[u]$ for the term where the hole has been replaced by u. Any term has at most one hole. We borrow the notions of weak and strong quantifiers [1]: since we are working in a refutation context, a positive occurrence of a quantifier $\exists\bar{x}.\,\varphi$ or a negative occurrence of a quantifier $\forall\bar{x}.\,\varphi$ is strong and a negative occurrence of a quantifier $\exists\bar{x}.\,\varphi$ or a positive occurrence of a quantifier $\forall\bar{x}.\,\varphi$ is weak. We will call the subformula ψ of $Q\bar{x}.\psi$, where $Q \in \{\forall, \exists\}$, the *matrix*, even though ψ might not be in clausal normal form. Without loss of generality, we assume that all quantified variables have been renamed to be distinct.

To handle strong quantifiers we use the Skolemization operator $\mathbf{sk}$. For a formula $Q\bar{x}.\psi$ it is defined as $\mathbf{sk}(Q\bar{x}.\psi, \bar{y}) := \psi[s_1(\bar{y})/x_1] \ldots [s_n(\bar{y})/x_n]$ and each s_i is a fresh function symbol of correct arity. The strong quantifier Q might not be below a weak quantifier. In this case we write $\mathbf{sk}(Q\bar{x}.\psi, \emptyset)$ and the fresh symbols are constants.

2.2 Preprocessing

Given an input formula $\mathcal{P}$ (i.e., a term of sort Bool) a CDCL($\mathcal{T}$) solver performs multiple preprocessing steps before the solving phase is started. This produces an equisatisfiable problem $\mathcal{P}'$ which is a conjunction of clauses.

To make efficient use of the ground solver, ground formulas are clausified. Quantified formulas, however, are treated differently. Strong quantifiers are usually not fully Skolemized, nor are the formulas put in prenex form or clausified. This has the benefit that the original structure of quantified formulas is preserved, which is crucial for some instantiation techniques.

Preprocessing applies some light form of rewriting on quantified formulas. In veriT, most rewriting steps apply to constants below arithmetic operators and Boolean connectives. For example, the term $f(5+c_1+3, c_2*(3-3))$ is simplified to $f(8+c_1, 0)$ and $(\bot \rightarrow \varphi_1) \rightarrow \varphi_2$ is replaced by φ_2. Rewriting also ensures that certain global invariants of veriT are met: for instance, all occurrences of bound variables are renamed to distinct variables, and quantifiers over Boolean variables are removed by Shannon expansion.

Another preprocessing step on quantified formulas is Skolemization. How Skolemization is applied is implementation dependent. The common CDCL($\mathcal{T}$) calculus [5] is only concerned with ground reasoning. While the SMT solver Z3 [15] has a builtin tactic called *nnf* that fully applies Skolemization, CVC4 [3] and veriT only Skolemize outermost strong quantifiers in their default configuration. The rewriter of veriT Skolemizes outermost strong quantifiers by replacing the subformula $\mathbf{trim}(\ell)$ of a formula ℓ by $\mathbf{sk}(\mathbf{trim}(\ell), \emptyset)$ if $\mathbf{trim}(\ell)$ has the form $Q\bar{x}.\,\varphi$, the quantifier Q is strong, and ℓ does not occur below any quantifier.

Due to the limited preprocessing of quantified formulas, some disjuncts of P' start with a weak quantifier and contain complicated formulas. To clarify the distinction, we call the disjuncts which start with a quantifier, and are hence black boxes to the ground solver, *boxes*. Without loss of generality, we assume all boxes are universally quantified. Disjuncts which are not boxes are ground literals. A *unit-box* is a clause with only one disjunct that is a box.

Example 2. The preprocessor will not perform any operations on Example 1. We can illustrate the perspective of the ground solver by replacing quantified formulas by frames. The resulting claues are: $\boxed{\ 1\ }$, $\neg P(c) \vee \boxed{\ 2\ }$, and $P(c)$. The first clause is a unit-box and both boxes will be abstracted to different propositional variables for the SAT solver.

2.3 Instantiation Techniques

The instantiation loop (Figure 1) starts with the *ground solver*. It either determines that the ground literals of the preprocessed problem $\mathcal{P}'$ are unsatisfiable or produces a *ground model* $\mathcal{M}$. If the ground problem is unsatisfiable, then $\mathcal{P}$ is unsatisfiable. $\mathcal{M}$ is a set of formulas $\mathcal{G} \cup \mathcal{Q}$ where $\mathcal{G}$ are ground literals, and $\mathcal{Q}$ are boxes. $\mathcal{M}$ propositionally satisfies $\mathcal{P}'$, and $\mathcal{G}$ is consistent with respect to the used theories. The instantiation procedure will then generate lemmas $(\forall \bar{x}.\, \varphi) \rightarrow \varphi\sigma$ where $(\forall \bar{x}.\, \varphi) \in \mathcal{Q}$ and σ is a substitution of $\bar{x}$ with ground terms. The generated lemmas are added conjunctively to $\mathcal{P}'$ and the ground solver is called again.

We now give an overview of common instantiation techniques and illustrate why they cannot tackle Example 1 quickly. These techniques are presented in the order they are used by veriT: it first tries conflict-driven instantiation. If this fails, it will try trigger-based instantiation. Should this produce no instances, it will fall back to enumerative instantiation. Model-based quantifier instantiation is not implemented by veriT: it is crucial for satisfiability, but veriT focuses on proving unsatisfiability.

Conflict-Driven Instantiation. This method tries to find an instance that contradicts the ground model $\mathcal{M}$ in the theory of equality and uninterpreted functions (EUF) [2, 17]. Hence, it searches for a box $\forall \bar{x}.\, \varphi \in Q$ and a substitution σ such that $\mathcal{G} \wedge \varphi\sigma \vDash_{\mathrm{EUF}} \bot$. It returns the instance $\varphi\sigma$ or fails. It can also search for substitutions which solve multiple constraints simultaneously. Hence, this method can find a contradicting instance of a clause $\psi_1 \vee \cdots \vee \psi_n$ by solving $\mathcal{G} \wedge \psi_1\sigma \vDash_{\mathrm{EUF}} \bot, \ldots, \mathcal{G} \wedge \psi_n\sigma \vDash_{\mathrm{EUF}} \bot$, but all ψ_is must be quantifier-free.

Conflict-driven instantiation is very helpful, since it only generates instances that are immediately useful. It forces the ground solver to find new models and eliminates spurious models from the search space.

Since assertion 2 of Example 1 contains a quantifier, it cannot be instantiated by conflict driven instantiation. Conflict driven instantiation also fails for assertion 1, because initially there is no ground formula that would be in conflict with an instance of $P(f(x, c))$. Even if the second assertion was Skolemized, conflict-driven instantiation would fail: since there is no ground instance of the Skolem term, no conflicting instance can be found.

Trigger-Based Instantiation. This instantiation scheme works by matching *triggers* with the current ground model. Triggers associate with every box $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ one or more lists of quantifier-free terms $t_1, \dots, t_n$ such that $\mathbf{free}(t_1) \cup \cdots \cup \mathbf{free}(t_n) = \{\bar{x}\}$. The triggers are either provided by the user or are heuristically generated. Trigger inference uses the structure of the quantified formulas which is preserved by preprocessing. To construct instances of φ, trigger-based instantiation searches for substitutions σ and terms $g_1, \dots, g_n \in \mathcal{T}(\mathcal{G})$ such that $\mathcal{G} \vDash_{\mathrm{EUF}} t_i\sigma \approx g_i$. If the search is successful, it returns the instance $\varphi\sigma$.

The process of matching terms within the theory of equality and uninterpreted functions is called *E-matching* [8, 10, 13]. Due to the heuristic nature of trigger-based instantiation, the generated instances might not be useful to solve the problem. Instead they can slow down or mislead the solver.

In the case of Example 1, a trigger $P(x)$ on assertion 1 would produce the useless instance $P(c) \rightarrow P(f(c,c))$ and a trigger $P(f(x,c))$ initially cannot match anything. The trigger $P(y)$ on assertion 2 would produce the instance $(\forall z.\, P(z) \rightarrow P(f(z,c))) \rightarrow \neg P(c)$. This instance is a step towards solving the problem: the strong variable z is no longer below a quantifier and will be Skolemized to create the formula $P(s_1) \rightarrow P(f(s_1,c)) \rightarrow \neg P(c)$ where s_1 is a fresh constant. During the next instantiation round the trigger $P(x)$ on assertion 1 generates the instance $P(s_1) \rightarrow P(f(s_1,c))$ which leads to the contradiction.

This technique is very sensitive to the availability of the right ground terms in the ground model. In the above example, if the formula contained $\forall x.\, P(x)$ instead of $P(c)$, trigger-based instantiation would have been helpless.

Enumerative Instantiation. While conflict driven instantiation is guided by the ground model it tries to contradict, and trigger-based instantiation is guided by the triggers, enumerative instantiation [16] is unguided. For a box with the form $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ it creates all substitutions $[\bar{t}/\bar{x}]$ where the terms $\bar{t}$ are ground terms from $\mathcal{T}(\mathcal{M})$. To limit the number of generated instances the procedure only uses the ground terms minimal with respect to some term order and does not return instances already implied by the ground model (i.e., it only returns $\varphi\sigma$ if $\mathcal{G} \nvDash_{\mathrm{EUF}} \varphi\sigma$). Enumerative instantiation ensures the theoretical completeness of the SMT solver for the theory of uninterpreted functions. It can also find the small ground terms that are sometimes necessary to enable the two previous techniques to work, and is thus a useful fallback strategy.

For Example 1, enumerative instantiation also needs at least two rounds. First, the variable y of assertion 2 is instantiated with c. Then, after Skolemization, assertion 1 can be instantiated with the new Skolem constant. Eventually, the cooperation of enumerative instantiation and the above techniques would succeed. However, in presence of many ground terms of the same sort as c, enumerative instantiation might have needed a lot of time to find the right instance.

Model-Based Quantifier Instantiation. Finally, model-based quantifier instantiation [11] extends heuristically the ground model $\mathcal{M}$ to a first-order interpretation $\mathfrak{I}$ and tests if this interpretation is a model: if there exists a box of the form $\forall \bar{x}.\, \varphi \in \mathcal{Q}$ and a substitution σ of $\bar{x}$ with terms from $\mathcal{T}(\mathcal{G})$ such that $\mathfrak{I} \vDash \neg\varphi\sigma$, then $\mathcal{M}$ is not a true model of $\mathcal{P}'$ and the ground instance $\varphi\sigma$ is produced. If this is not the case, then $\mathfrak{I}$ is a model of $\mathcal{P}'$ and the input problem is satisfiable. This methods works for every fragment that has the finite model property.

For Example 1, model-based quantifier instantiation fails to generate the right instances in one round for the same reason that trigger-based and enumerative instantiation fail: it might instantiate assertion 2 with c for y, but other rounds of instantiations will still be required to reach a contradiction.

3 Quantifier Simplification by Unification

The essence of our technique is to simplify boxes by replacing a quantified subformula of the box with the Boolean constant $\top$ or $\bot$. This can be done if the matrix of this quantified subformula can be unified with the matrix of a unit-box.

Example 3. On our running Example 1, the first assertion serves as unit-box, whose matrix is unifiable with the matrix of the box in the second assertion. As a result, the quantified subformula can be reduced to the Boolean constant $\top$, for some instance of the second formula.

$$\frac{\forall x.\, P(x) \rightarrow P(f(x,c)) \quad \forall y.\, (\forall z.\, P(z) \rightarrow P(f(z,y))) \rightarrow \neg P(y)}{\top \rightarrow \neg P(c)}$$

The rewriter simplifies the formula $\top \rightarrow \neg P(c)$ to $\neg P(c)$. Notice that, in this example, the variable z must be Skolemized because its quantifier is strong.

The SUB rule (Section 3.1) formalizes this derivation. An SMT solver can use this rule to augment the problem with simplified formulas. It is carefully restricted to generate formulas which help the instantiation procedures (Section 3.2). In Section 3.3 we propose several variants of the rule with different tradeoffs.

3.1 The Core Rule

The simplification by unification of subformulas (SUB) rule simplifies a box by replacing a quantified subformula with a Boolean constant. To be able to do so, the rule unifies the matrix of the subformula with a unit-box using a substitution. The Boolean constant depends on the polarities of the matrices: if they have the same polarity the subformula is replaced by $\top$, if they have different polarity it is replaced by $\bot$. The conclusion of the rule is the *pre-simplified* formula and will be fully simplified by the rewriter.

Definition 1 (SUB Rule).

$$\frac{\forall x_1, \ldots, x_n.\, \psi_1 \quad \forall x_{n+1}, \ldots, x_m.\, \varphi[Q\bar{y}.\, \psi_2]}{\forall x_{k_1}, \ldots, x_{k_j}.\, \varphi[b]\sigma}\,\text{SUB}$$

where $Q \in \{\exists, \forall\}$, the subformula $Q\bar{y}.\, \psi_2$ appears only below the outermost universal quantifier of φ, and σ is a substitution. The rule is subject to the conditions:

1. *$\mathbf{trim}(\psi_1)\sigma = \mathbf{trim}(\psi_2)\sigma$, if $Q\bar{y}.\,\psi_2$ is weak;*
2. *$\mathbf{trim}(\psi_1)\sigma = \mathbf{trim}(\mathbf{sk}(Q\bar{y}.\,\psi_2, x_{n+1} \dots x_m))\sigma$, if $Q\bar{y}.\,\psi_2$ is strong;*
3. *The bound variables of the conclusion $\{x_{k_1}, \dots, x_{k_j}\}$ are exactly $\mathbf{free}(\varphi[b]\sigma)$;*
4. *$b = \top$ if $\mathbf{pol}(\psi_1) = \mathbf{pol}(\psi_2)$ or $b = \bot$ if $\mathbf{pol}(\psi_1) \neq \mathbf{pol}(\psi_2)$.*

Example 4. In the running example the subformula $\forall z.\, P(z) \to P(f(z, y))$ occurs negatively. Since $Q = \forall$, the formula must be Skolemized (Condition 2):

$$\mathbf{sk}(\forall z.\, P(z) \to P(f(z, y)), y) = P(s_1(y)) \to P(f(s_1(y), y))$$

Hence, the unifier used in Example 3 is $\sigma = [s_1(c)/x][c/y]$.

Example 5. Ignoring Skolemization (Condition 2) leads to unsoundness:

$$\frac{\forall x.\, P(x, x) \quad \forall y.\, \neg(\forall z.\, P(y, G(z)))}{\neg\top}$$

The result of Skolemization $\mathbf{sk}(\forall z.\, P(y, G(z)), y)) = P(y, G(s_1(y)))$ is not unifiable with $P(x, x)$. The rule is not applicable.

Example 6. The conclusion can contain variables from both premises. Here the unifier is $\sigma = [G(x)/y_1][c/z]$.

$$\frac{\forall x.\, P(G(x), c) \quad \forall y_1, y_2.\, (\forall z.\, P(y_1, z)) \wedge P(y_1, y_2)}{\forall x, y_2.\, \top \wedge P(G(x), y_2)}\,\text{SUB}$$

Example 7. The above examples were cases where $\mathbf{pol}(\psi_1) = \mathbf{pol}(\psi_2)$. This example illustrates the other case:

$$\frac{\forall x.\, \neg P(x, x) \quad \forall y.\, G(c) \wedge (\forall z.\, P(y, z))}{G(c) \wedge \bot}\,\text{SUB}$$

In this case, the rewriter will simplify the pre-simplified formula $G(c) \wedge \bot$ to $\bot$ and the SMT solver can directly deduce unsatisfiability.

The SUB rule allows us to simply combine and restrict Skolemization, unification, and the replacement of subformulas with the appropriate constant. In the next section we will see the role it has within an SMT solver. The rule soundly combines these sound steps. First, it Skolemizes the variables $\bar{y}$. Second, it applies the unifier σ. Now the subformula of ψ_1 corresponding to $Q\bar{y}.\,\psi_2$ in the SUB rule is equivalent to $\mathbf{trim}(\psi_1)\sigma$ and is replaced with a Boolean constant. The constant is chosen appropriately according to the polarity of the formulas. This replacement is sound since $\psi_1\sigma$ always holds. Overall, the SUB rule, together with applying the rewriter, somewhat resembles unit resolution where $\forall x_1, \dots, x_n.\, \psi_1$ is the unit clause. In the case of SMT solvers, however, φ might not be a clause. Furthermore, ψ_1 and ψ_2 will have the complex structure that is preserved from the input, since most currently used instantiation techniques have no advantage from applying full clausification.

3.2 The Simplification Within the SMT Solver

Since the SUB rule eliminates a quantified subformula, the conclusion is easier to handle for the SMT solvers. In general, however, the conclusion does not subsume the box serving as the second premise. Hence, this box cannot simply be replaced by the conclusion. Instead, the problem must be augmented with the derived box. As the evaluations show, augmenting the problem still helps the SMT instantiation procedures to find the appropriate ground instances.

```
   $I \leftarrow \emptyset$
   $Q$ is an empty queue.
   for each clause $\mathcal{C}$ in $\mathcal{P}'$ do
4      if $\mathcal{C}$ is unit-box with the box $\ell$ then $I \leftarrow I \cup \{\ell\}$
5      if $\mathcal{C}$ contains a box then push$(Q, \mathcal{C})$
   while $Q$ is not empty do
       $\ell_1 \vee \cdots \vee \ell_n \leftarrow$ pop$(Q)$
8      if there is $\psi \in I$ and a box $\ell_i$ such that $\dfrac{\psi \quad \ell_i}{\ell'}\,\textsc{sub}$ then
9          $\ell' \leftarrow$ rewrite$(\ell')$
10         $\mathcal{C}' \leftarrow \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell' \vee \ell_{i+1} \vee \cdots \vee \ell_n$
11         append $\mathcal{C}'$ to $\mathcal{P}'$
           if $\mathcal{C}'$ contains a box and is not an unit-box then
13             push$(Q, \mathcal{C}')$
```

Fig. 2. The augmentation procedure.

The pseudocode in Figure 2 shows the loop which augments the problem. It is executed after preprocessing finishes and before the ground solver starts. The procedure first iterates over the clauses in the preprocessed problem $\mathcal{P}'$ to build a set I of unit-boxes (Line 4) which can be used to simplify quantified subformulas. At the same time, this loop collects in a queue Q all clauses containing boxes (Line 5). Then the procedure takes a clause from the queue and tries to simplify one of its boxes. To do so, it uses the SUB rule. If this succeeds, the conclusion is the pre-simplified formula. The procedure then uses the rewriter to finish the simplification and the problem is augmented with the simplified formula by adding it conjunctively to the problem (Lines 8 to 11). If the simplified clause still contains a box, it is pushed back onto the queue (Line 13).

The procedure terminates since the queue Q will eventually be empty. Every iteration removes a clause from the queue and adds at most one new clause. When the test in line 8 fails, i.e., the SUB rule can not be applied, no new clause is added. Otherwise, it adds a clause with fewer nested formulas that can serve as $Q\bar{y}.\psi_2$ in the SUB rule. Hence, the SUB rule will eventually no longer apply to any box left in the clauses in Q.

The approach of augmenting the problem with derived, but new, formulas bears the risk that the instantiation procedures create more useless ground instances from the new formulas. To minimize this risk, the SUB rule is restricted to only apply when the result is likely to be helpful. First, the detection of subformulas which can be eliminated only uses unification instead of a more general approach. Since preprocessing preserves the structure of quantified formulas, unifiability can indicate the intention of the user. For example, the unit-box might be a lemma that is used within the box that is simplified. Second, the first premise must be a box. In principle it could also be a ground literal, but ground literals are already directly usable by the ground solver. Third, the simplified subformula must start with a quantifier because the instantiation procedures struggle to instantiate the quantified subformula. One of the variants described in the next section drops this restriction, but it is not as useful as the restricted rule.

3.3 Variants

As the experimental evaluation (Section 5) shows, the above version of quantifier simplification by unification solves more instances at little cost. Nevertheless, we also developed several variants with different tradeoffs. We will call quantifier simplification by unification as presented so far the *normal variant* and will often drop the phrase *by unification* to avoid repetition.

Eager Simplification. Since quantified subformulas block the instantiation procedures from creating the right instances quickly, the SUB rule is restricted to only simplify quantified subformulas. This restriction, however, can be removed to generate more simplified formulas. The eager SUB rule is the rule

$$\frac{\forall x_1, \dots, x_n.\, \psi_1 \quad \forall x_{n+1}, \dots, x_m.\, \varphi[\psi_2]}{\forall x_{k_1}, \dots, x_{k_j}.\, \varphi[b]\sigma}\ \text{eager-SUB}$$

and all side conditions of SUB are changed to read ψ_2 in-place of $Q\bar{x}.\, \psi_2$.

The eager SUB rule can be applied on any subformula not below an extra quantifier. On the one hand, this corresponds to deriving general consequences of unit-boxes in full first-order logic, but on the other hand, it will generate many more new formulas which potentially slow down or misguide the solver.

Solitary Variable Heuristic. To limit the potential downsides of eager simplification, we can limit the cases when the rule is applied: we apply the rule when it potentially removes a variable from the outermost quantifier of the second premise. The resulting formula will produce fewer misleading instances.

A variable is removed from the pre-simplified formula if it is *solitary*: it appears in the subformula ψ_2, but not in any other subformula of φ. Hence, for example, in the case $\varphi = t_1 \vee \cdots \vee t_i \vee \cdots \vee t_n$ we apply the rule with $\psi_2 = t_i$ if there is a variable $x \in \mathbf{free}(t_i)$ such that $x \notin \mathbf{free}(t_1 \vee \cdots \vee t_{i-1} \vee t_{i+1} \vee \cdots \vee t_n)$.

Deletion of Simplified Clauses. Another way to restrict the number of newly created instances is to delete the clause that contains the box used as the second premise of the SUB rule after it has been simplified. While this is no longer complete, it can guide the solver towards solving the refutation problem. Especially, within a strategy schedule this can be a valuable strategy.

This variant can be combined with the three other variants. Overall, this results in six variants of quantifier simplification. The amount of clauses deleted depends on the activity of the simplification variant used. Especially in the case of eager simplification with deletion many input assertions will be deleted.

4 Implementation

Our implementation of quantifier simplification by unification in veriT uses a non-perfect discrimination tree as term index and a subsequent unifiability check (Section 4.1). Both steps are amended to take strong variables into account without explicit Skolemization and avoid the creation of unnecessary Skolem symbols.

The implementation also does not apply the simplification of clauses everywhere, but focuses on unit-boxes only: the queue Q will only be populated by unit-boxes. This simplifies the implementation, since we do not have to track which boxes of a clause have already been simplified. It indeed appears that in SMT-LIB benchmarks clauses with boxes are uncommon and quantified formulas are usually unit-boxes (e.g., quantifiers range over entire disjunctions). A prototype without this simplification did not perform better on these benchmarks than the simplified version.

4.1 Indexing and Unification Without Skolemization

A key element to execute quantifier simplification, as shown in the algorithm in Figure 2, is the lookup of the unit-box $\forall \bar{x}.\, \psi_1$ from the index I. The trimmed matrix of this box must be unifiable with the trimmed matrix of the quantified subformula $Q\bar{y}.\, \psi_2$. To implement the search for unifiable formulas efficiently we use a term index. We use non-perfect discrimination trees [20]. Non-perfect means that the lookup is an over-approximation: some returned terms are not unifiable with the query term and must be removed by a full unification step.

For each unit-box $\forall \bar{x}.\, \psi_1$ (of I in the algorithm in Figure 2) the index stores $\mathbf{trim}(\psi_1)$ together with $\mathbf{pol}(\psi_1)$. For each possible subformula $Q\bar{y}.\, \psi_2$ the implementation uses $\mathbf{trim}(\psi_2)$ as a query term and retrieves unification candidates and their polarity. Afterwards, it performs a full unification to construct the substitution σ when possible. If, however, the quantifier of $Q\bar{y}.\, \psi_2$ is strong, the subformula should be Skolemized.

To handle variables that would be replaced by Skolem terms, the lookup process is enhanced: while normal variables are replaced by a variable placeholder that can match any term, variables to be Skolemized act like constants and can not match any other term. This embeds Skolemization into indexing, since Skolem terms start with fresh function symbols that can never match indexed terms.

After the index returns a filtered set of possible premises $\mathbf{trim}(\psi_1)$ with their polarities $\mathbf{pol}(\psi_1)$ from I, the implementation must use full unification [18] to eliminate false positives and to build the unifier σ. It has to solve the unification problem between $\mathbf{trim}(\psi_1)$ and $\mathbf{trim}(\psi_2)$, where $\mathbf{trim}(\psi_2)$ can contain variables that must be Skolemized.

To handle Skolemized variables during unification, our implementation deviates from the standard version in two ways. First, similarly to the term index, it handles Skolemized variables as constants. Second, it considers a Skolemized variable as an occurrence of all the variables its Skolem term would depend on.

The resulting unifier σ cannot substitute a Skolem term into the quantified variables $x_{n+1}, \ldots, x_m$ of the box that is simplified. Hence, the conclusion $\varphi[b]\sigma$ is free of any Skolem terms, and no Skolem term has ever to be constructed. Overall, restricting quantifier simplification by unification to not simplify formulas below multiple nested quantifiers allows for this elegant implementation.

5 Evaluation

This section presents an empirical evaluation of quantifier simplification by unification and its variants as implemented in veriT.[3] The default variant of quantifier simplification solves more benchmarks than the default configuration of veriT, while losing few benchmarks. This justifies the activation of our quantifier simplification method in the default configuration. Almost all other variants also solve more benchmarks than the default configuration. veriT exposes a wide range of options to fine-tune the instantiation module. A specific configuration is a *strategy*. Quantifier simplification solves benchmarks not solved by any veriT strategy without this technique (Section 5.1).

In order to fully benefit from the strategies available, veriT can use strategy schedules. We generated strategy schedules with and without quantifier simplification and evaluated their performance. The strategies with quantifier simplification are an integral component of the generated schedules and increase the number of solved benchmarks. They are especially useful for short timeouts (Section 5.2).

We performed the experiments on the benchmarks from the SMT-LIB benchmark release 2021 [4]. Since quantifier simplification is only relevant for first-order formulas, we used the SMT-LIB logics supported by veriT which use quantifiers, uninterpreted functions, or arrays. Those are the SMT-LIB logics UF, UFLRA, UFLIA, UFIDL, ALIA, AUFLIA, and AUFLIRA. Since veriT is purely refutational, we removed benchmarks known to be satisfiable from the analysis.[4] Overall, the SMT-LIB contains 41 129 benchmarks using these logics. Of those, 1206 benchmarks are known to be satisfiable. This leaves 39 923 relevant benchmarks. We used the 2021.06-rmx release of veriT.

To interpret the numbers, the reader should keep in mind that veriT has no array solver. It treats the functions of the SMT-LIB theory of arrays as uninterpreted functions. Since veriT is restricted to refute benchmarks, this

[3] The raw data is available on Zenodo [**?**].

[4] Benchmarks known to be satisfiable can identify soundness problems. Hence, we included them in the experiments, but removed them from the data.

Table 1. Comparison with the default strategy and the theoretical best solver on 39 923 benchmarks.

vs. Default (solves 31 690)	N	E	S	Nd	Ed	Sd	Total
Solved	31 927	31 772	31 928	31 733	21 405	21 823	32 151
	+237	+82	**+238**	+43	−10 285	−9 867	+461
Gained	282	**315**	285	291	115	255	475
Lost	**45**	233	47	248	10 400	10 122	14
vs. Theoretical Best (solves 32 633)							
Gained	83	80	85	**86**	32	76	**125**

approach is sound. Nevertheless, veriT can fail to solve easy benchmarks that require array reasoning.

All experiments have been performed on computers with one Intel Xeon Gold 5220 processors with 18 cores and 96 GiB RAM. We ran one instance of veriT per available core and used a memory limit of 6 GiB per instance.

5.1 Baseline Comparison

Table 1 shows the number of benchmark solved within a timeout of 180 s in comparison to the default strategy. The standard version of quantifier simplification by unification is denoted *N*, eager simplification is denoted *E*, and the solitary variable heuristic is denoted *S*. A suffix *d* denotes the deletion of simplified clauses. Benchmarks are "Gained" if they are not solved by the default strategy and "Lost" if they are solved by the default strategy, but not by the variant. The column "Total" reports the union of the benchmarks solved by all variants.

The normal variant shows a good improvement by solving 237 benchmarks more. Most other variants solve more benchmarks not solved by the default strategy, but also lose many more. While the normal variant does not have the highest gain, the small loss justifies enabling it in the default strategy of veriT. The huge number of lost benchmarks for the variants that combine eager simplification and the solitary variable heuristic with clause deletion is not surprising: since most input assertions can be simplified in some way, clause deletion removes much of the original problem. The result is often an unsolvable problem.

Compared to the union of benchmarks solved by *any* existing veriT strategy (Theoretical Best), quantifier simplification by unification shows good improvement. We used a list of 43 strategies which are also used by veriT in the SMT competition.[5] The default configuration of veriT is on this list. Overall, the variants together are able to solve 125 benchmarks that veriT could not solve before. While eager simplification solves only 80 more, 18 of those are not solved by any other quantifier simplification variant. Here, the two variants with clause deletion that have a huge loss are somewhat redeemed: together they solve eight benchmarks not solved by the theoretical best solver and the other quantifier simplification variants.

[5] Competition website: `https://smt-comp.github.io/`

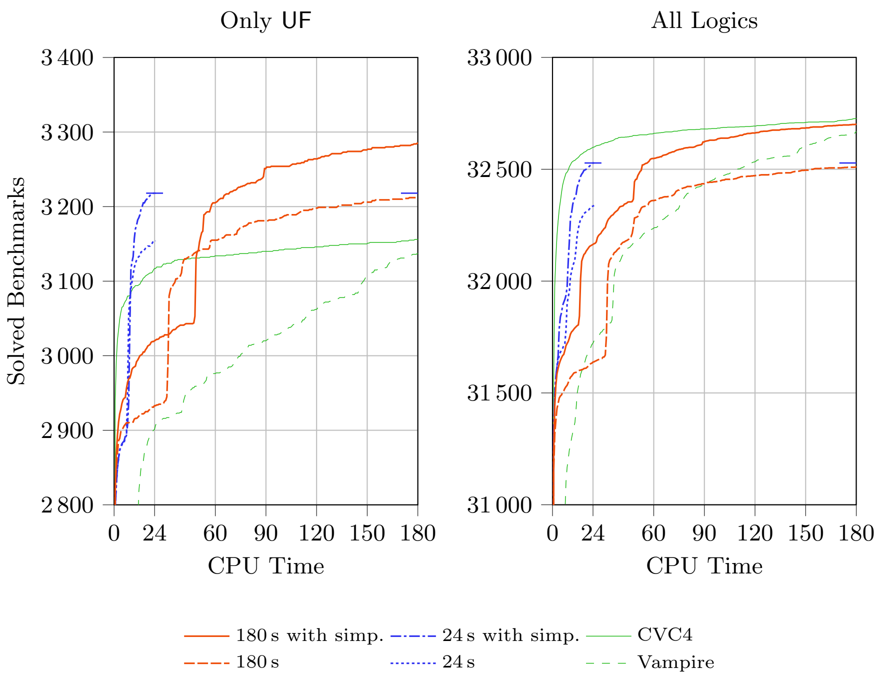

Fig. 3. CDF of different schedules on UF only and all logics.

To perform the quantifier simplification, veriT does not need much time: for the normal variant, we measured a median runtime of 0.5 ms and mean of 3 ms.

5.2 Strategy Scheduling

Since quantifier instantiation relies on heuristics, veriT exposes parameters that can be set by the user in a strategy. Most benchmarks are solved by an appropriate strategy within a short timeout. Hence, it is sensible to execute many strategies for short time intervals one after another in a schedule.

To evaluate the quantifier simplification technique within a strategy schedule, we generated schedules with and without strategies extended with quantifier simplification. An optimal schedule is the set of strategy–timeout pairs which solves the most benchmarks. The list of possible strategies and timeouts is handcrafted. We use integer programming to solve this optimization problem. veriT itself uses the logic of the problem to select a schedule.

To build strategies with quantifier simplification we picked six strategies from the strategy list of 43 strategies: the default strategy, the strategy that solved the most benchmarks overall, and four complementary strategies. The four complementary strategies were selected by finding a pair of strategies that together with the best strategy maximize the number of solved benchmarks. We searched such a pair on all logics and on first-order logic with equality (UF) alone. We then extended these six strategies with the six variants of quantifier simplification. This resulted in 36 new strategies.

We generated schedules optimized for timeouts of 180 s and 24 s. The short 24 s timeout allows us to evaluate the value of quantifier simplification for applications such as interactive theorem provers, which require a short timeout. It corresponds to the timeout used by the SMT competition to evaluate solvers for this purpose. The longer 180 s timeout was arbitrarily chosen.

Figure 3 shows the number of benchmarks solved within a time limit on UF alone and on all logics. On all logics, the schedule with quantifier simplification solved 193 benchmarks more after 24 s than the original 24 s schedule. For the 180 s timeout, the 180 s schedule with quantifier simplification solves 191 more than the one without. The 24 s schedule with quantifier simplification solves 18 benchmarks more than the 180 s schedule after 180 s. Hence, quantifier simplification is very useful for short timeouts. Since the form of quantified lemmas that quantifier simplification by unification eliminates appear in problems generated by interactive theorem provers, it is especially useful for this application.

To provide context the plots contain the results of two other systems: the state-of-the-art SMT solver CVC4 and the superposition prover Vampire [12]. We used the official builds of version 1.8 of CVC4 and version 4.5.1 of Vampire. Vampire includes the SMT solver Z3, which aids theory reasoning. Since CVC4 has no scheduler optimized for 24 s or 180 s, we ran the default strategy.[6] For Vampire we used the SMT-COMP scheduler with a timeout of 180 s.[7] We discarded all "satisfiable" results. Overall, CVC4 solves 70 benchmarks more than veriT with quantifier simplification after 24 s and 26 after 180 s. veriT with quantifier simplification after 180 s solves 595 benchmarks not solved by CVC4, of which 107 are also not solved by veriT without quantifier simplification. Surprisingly, Vampire solves fewer benchmarks than any other system on UF. This is due to the nature of typical SMT benchmarks: they usually require little quantifier reasoning and are hence easier to solve for instantiation-based systems.[8] This confirms that restricted methods, such as quantifier simplification by unification, are useful for SMT problems.

Figure 4 visualizes the schedules for the logic UF. Grey cells are strategies that use quantifier simplification. Cells with the same number use the same base strategy. Some base strategies appear both in the schedules with and without quantifier simplification. The strategies with quantifier simplification tend to be used for shorter time slices than the variants without.

[6] Using: `-L smt2.6 --no-incremental --no-type-checking --no-interactive --full-saturate-quant`

[7] Using: `-t 180s -m 6000 --mode portfolio --schedule smtcomp --input_syntax smtlib2 -om smtcomp -p off`

[8] This has been confirmed to us by the Vampire team in conversations.

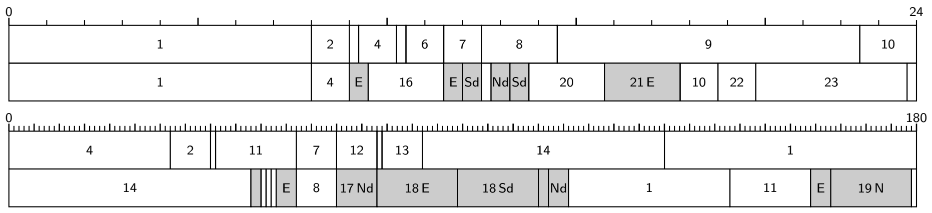

Fig. 4. Visualization of optimized UF schedules. The bottom rows are the schedules with quantifier simplification. The numbers denote the base strategies.

6 Conclusion

We presented a new unification-based simplification technique for instantiation-based SMT solvers. Its design is motivated by limitations of modern instantiation methods, and it is efficient. Problems where formulas can be simplified are often solved much faster, despite the method creating new quantified formulas. We plan to enable quantifier simplification by unification by default in the next veriT release. The release will also produce machine-checkable proofs for simplifications performed by quantifier simplification by unification.

We believe that the technique implemented here within veriT can be ported easily into any instantiation-based SMT solver, and we are confident that it would also enable mainstream solvers to tackle problems outside of reach with other current strategies. We will investigate its potential in other solvers.

Our method is a step towards using techniques inspired by resolution-based theorem provers within SMT solvers. It is currently only used as a preprocessing technique, but we plan to investigate novel quantifier instantiation techniques which can directly handle nested strong quantifiers.

Acknowledgments. We are grateful to Haniel Barbosa, Jasmin Blanchette, Daniel El Ouraoui, Mathias Fleury, Martin Riener, and Athénaïs Vaginay for many fruitful discussions and suggestions to improve the text. We thank the anonymous reviewers for many good suggestions to improve the text. The second author has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

1. Baaz, M., Egly, U., Leitsch, A., Goubault-Larrecq, J., Plaisted, D.: Chapter 5 - normal form transformations. In: Robinson, A., Voronkov, A. (eds.) Handbook of

Automated Reasoning, pp. 273–333. Handbook of Automated Reasoning, North-Holland, Amsterdam (2001). https://doi.org/10.1016/B978-044450813-3/50007-2
2. Barbosa, H., Fontaine, P., Reynolds, A.: Congruence closure with free variables. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 214–230. Springer Berlin Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_13
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
5. Barrett, C.W., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 305–343. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_11
6. Bonacina, M.P., Lynch, C., de Moura, L.: On deciding satisfiability by theorem proving with speculative inferences. Journal of Automated Reasoning **47**, 161–189 (08 2011). https://doi.org/10.1007/s10817-010-9213-y
7. Bouton, T., de Oliveira, D.C.B., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 22. LNCS, vol. 5663, pp. 151–156. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_12
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52**(3), 365–473 (May 2005). https://doi.org/10.1145/1066100.1066102
9. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.: SMTCoq: A plug-in for integrating SMT solvers into Coq. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 126–133. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_7
10. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 21. pp. 167–182. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_12
11. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. pp. 306–320. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_25
12. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. pp. 1–35. Springer Berlin Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
13. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 21. pp. 183–198. Springer Berlin Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_13
14. de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) Automated Reasoning. pp. 475–490. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_40
15. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer Berlin Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24

16. Reynolds, A., Barbosa, H., Fontaine, P.: Revisiting enumerative instantiation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 112–131. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-89963-3_7
17. Reynolds, A., Tinelli, C., de Moura, L.: Finding conflicting instances of quantified formulas in smt. In: FMCAD 2014. pp. 195–202. IEEE (2014). https://doi.org/10.1109/FMCAD.2014.6987613
18. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM **12**(1), 23–41 (Jan 1965). https://doi.org/10.1145/321250.321253
19. Schurr, H.J., Fleury, M., Desharnais, M.: Reliable reconstruction of fine-grained proofs in a proof assistant. In: Platzer, A., Sutcliffe, G. (eds.) CADE 28. pp. 450–467. LNCS, Springer International Publishing, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5_26
20. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term Indexing, p. 1853–1964. Elsevier Science Publishers B. V., Amsterdam, Netherlands (2001). https://doi.org/10.5555/778522.778535
21. Voronkov, A.: AVATAR: The architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. pp. 696–710. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-08867-9_46