A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover

K. Rustan M. Leino [0], Madan Musuvathi [0], and Xinming Ou [1]

[0] Microsoft Research, Redmond, WA, USA
{leino,madanm}@microsoft.com
[1] Princeton University, Princeton, NJ, USA
xou@cs.princeton.edu

Abstract. Lazy proof explication is a theorem-proving architecture that allows a combination of Nelson-Oppen-style decision procedures to leverage a SAT solver's ability to perform propositional reasoning efficiently. The SAT solver finds ways to satisfy a given formula propositionally, while the various decision procedures perform theory reasoning to block propositionally satisfied instances that are not consistent with the theories. Supporting quantifiers in this architecture poses a challenge as quantifier instantiations can dynamically introduce boolean structure in the formula, requiring a tighter interleaving between propositional and theory reasoning.

This paper proposes handling quantifiers by using two SAT solvers, thereby separating the propositional reasoning of the input formula from that of the instantiated formulas. This technique can then reduce the propositional search space, as the paper demonstrates. The technique can use off-the-shelf SAT solvers and requires only that the theories are checkpointable.

0 Introduction

Automatic verification of hardware and software systems requires a good decision procedure for the conditions to be verified. Verification conditions generated for the verification of software involve functions and predicates for many types of values, including those of the source programming language. Designing decision procedures for these individual theories may be easier than designing a decision procedure that handles all of them. Nelson and Oppen [11, 10] developed a famous method for combining decision procedures of a class of first-order theories. Because of its modular architecture, theorem provers based on this method can readily support many interesting theories that are useful in practice. Many theorem provers are based on such combinations, for example Simplify [4], Verifun [6], ICS [2], and CVC Lite [1], and these have been applied to the verification of hardware and software systems.

Software verification conditions also involve quantified formulas. For example, the verification conditions generated by the program checker ESC/Java [8] use quantified formulas in several ways: (0) to specify a partial set of properties of otherwise uninterpreted functions, (1) to axiomatize properties guaranteed by Java and its type system, (2) to describe a procedure call's effect on the program heap, (3) to state object invariants for all objects of a class, and (4) to support quantifiers, usually over array elements, supplied by the user. Unfortunately, the Nelson-Oppen combination method is applicable only to quantifier-free first-order formulas. Reasoning about quantifiers in this setting cannot be handled as an ordinary theory, but instead needs special support. Another problem is that it is not always possible to have a terminating decision procedure when the input formulas contain quantifiers, but the prevalence of quantified formulas in important problems demands that the theorem provers handle them effectively in practice.

The Simplify theorem prover [4] provides support for quantified formulas that has been shown to be effective for software verification applications, for example in extended static checking [5, 8]. Simplify uses a kind of pattern matching of ground terms to trigger the instantiation of universally quantified formulas. However, Simplify does not handle propositional search very efficiently. A new generation of theorem provers, including Verifun [6], ICS [2], and CVC Lite [1], attempt to speed up the propositional search by leveraging the last decade's advances in SAT solving and using a *lazy-proof-explication* architecture. In such an architecture, a Nelson-Oppen combination of decision procedures interacts with an off-the-shelf SAT solver: the SAT solver finds ways to satisfy a given formula propositionally, while the combination of other decision procedures performs theory reasoning to block propositionally satisfied instances that are not consistent with the theories.

To use such a new-generation theorem prover in software verification applications, we seek to incorporate support for quantified formulas in the lazy-proof-explication architecture. This poses the following key challenges. First, quantified formulas typically involve propositional connectives. As a result, quantifier instantiations performed during theory reasoning can dynamically introduce boolean structure in the formula. This requires tighter interleaving between propositional and theory reasoning. Second, most quantifier instantiations are not useful in proving the validity of the formula. Blindly exposing such redundant instantiations to the SAT solver could drastically reduce the performance of the propositional search.

Support for quantified formulas in a lazy-proof-explication prover has been incorporated into Verifun [7]. When the quantifier instantiations result in formulas with propositional structure, Verifun augments the original formula with such instantiations so that the SAT solver can find ways to satisfy these instantiations in the context of the original formula. However, the added disjunctions then persist in the prover's state.

As an alternative approach, we propose a *two-tier* technique in this paper. This technique involves two off-the-shelf SAT solvers, a *main* solver that performs the propositional reasoning of the input formula, and a *little* solver that reasons over the quantifier instantiations. When the main SAT solver produces a propositionally satisfying instance that is consistent with the decision procedures, a pattern matching algorithm, similar to the one in Simplify, generates a set of quantifier instantiations. The little SAT solver, along with the decision procedures, tries to falsify the satisfying instance with the instantiations produced. If successful, the little SAT solver then generates a blocking clause that only contains literals from the input formula. By thus separating the propositional reasoning of the input formula from that of the instantiated formulas, this technique reduces the propositional search space.

Section 1 introduces some preliminaries and reviews the architecture of theorem provers based on lazy proof explication. Section 2 discusses the main problem in handling quantifiers in lazy-proof-explication theorem provers. The quantifier algorithm is presented in Section 3. We trace through an example in Section 4 and report on our preliminary experience with an implementation of the algorithm in Section 5. The final sections offer a discussion, some related work, and a conclusion.

1 Theorem proving using lazy proof explication

In this section, we review in more detail the architecture and main algorithm of a theorem prover based on lazy proof explication.

1.0 Terminology

A *formula* is constructed from an arbitrary combination of function and predicate symbols, propositional connectives, and first-order quantifier bindings. The following is an example formula:

$$\begin{aligned}&(\forall\, a, i, v \bullet\ 0 \leqslant i \wedge i < \mathit{Length}(a) \ \Rightarrow\ \mathit{read}(\mathit{write}(a, i, v), i) = v\ ) \wedge\\ &\mathit{Length}(b) > 0\\ &\ \Rightarrow\ \mathit{read}(\mathit{write}(b, 0, 10), 0) = 10\end{aligned}$$

An *atom* is a formula that does not start with a propositional connective. Propositional connectives include conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\Rightarrow$). For example, the following are all atoms:

$$\begin{aligned}&(\forall\, a, i, v \bullet\ 0 \leqslant i \wedge i < \mathit{Length}(a) \ \Rightarrow\ \mathit{read}(\mathit{write}(a, i, v), i) = v\ ),\\ &\mathit{Length}(b) > 0,\\ &\mathit{read}(\mathit{write}(b, 0, 10), 0) = 10.\end{aligned}$$

A *quantifier atom* is an atom that starts with a quantifier. A *literal* is either an atom or its negation. A *quantifier literal* is either a quantifier atom or its negation. A *monome* is a set of literals. If $\mathcal{P}$ is a set of formulas, we sometimes write just $\mathcal{P}$ when we mean the conjunction of the formulas in $\mathcal{P}$.

A theorem prover can be equivalently viewed either as a validity checker or a satisfiability checker: establishing the validity of a given conjecture P is equivalent to showing a satisfying assignment does not exist for $\neg P$. For the theorem provers discussed in this paper, we take the second view, thinking of the input as a formula (the negation of a conjecture) to be satisfied or shown unsatisfiable. We define three notions of satisfiability for a formula Φ: propositional satisfiability ($\mathit{PSat}(\Phi)$), satisfiability with theories ($\mathit{TSat}(\Phi)$), and satisfiability with theories and quantifiers ($\mathit{QSat}(\Phi)$).

0. $\mathit{PSat}(\Phi) = \mathit{True}$ if there exists a satisfying truth value assignment to every atom in Φ.
1. $\mathit{TSat}(\Phi) = \mathit{True}$ if $\mathit{PSat}(\Phi) = \mathit{True}$ and the truth value assignment to the non-quantifier atoms is consistent with the underlying theories.

2. $QSat(\Phi) = True$ if $PSat(\Phi) = True$ and the truth value assignment to the atoms is consistent with both the underlying theories and the semantics of quantifiers.

Proposition 0 *$QSat(\Phi)$ implies $TSat(\Phi)$, which in turn implies $PSat(\Phi)$.*

We define a *lemma* to be a formula that does not affect the satisfiability of any formula. That is, if P is a lemma and Φ is any formula, then $QSat(\Phi)$ iff $QSat(\Phi \wedge P)$. Note that if both P and Q are lemmas, then so is $P \wedge Q$. And if P is a lemma and P implies Q, then Q is also a lemma. In this paper, we use three kinds of lemmas:

0. a tautology generated by the theories,
1. a quantifier instantiation lemma of the form $(\forall x \bullet \ P(x)\ ) \Rightarrow P(a)$, which is also a tautology,
2. a quantifier skolemization lemma of the form $(\exists x \bullet \ P(x)\ ) \Rightarrow P(K)$ for an appropriate skolem function K, as defined later.

1.1 Lazy proof explication

In a lazy-proof-explication theorem prover, an off-the-shelf SAT solver conducts propositional reasoning of an input formula Φ. The SAT solver treats each atom in Φ as an opaque propositional variable. When possible, the SAT solver returns a truth value assignment m of these atoms that propositionally satisfies Φ. The theorem prover then invokes the theory-specific decision procedures to determine if the monome m is consistent with the underlying theories. If so, the input formula Φ is satisfiable. If not, the theories are responsible for producing a lemma that shows the monome m to be inconsistent. By conjoining this lemma to Φ—which by the definition of lemma does not change the satisfiability of Φ—the theorem prover blocks the assignment m.

For example, suppose a theorem prover is asked about the satisfiability of the following formula:

$$([\![x \leqslant y]\!] \vee [\![y = 5]\!]) \ \wedge \ ([\![x < 0]\!] \vee [\![y \leqslant x]\!]) \ \wedge \ \neg[\![x = y]\!]$$

where for clarity we have enclosed each atom within special brackets. As (the propositional projection of) this formula is passed to the SAT solver, the SAT solver may return a monome containing the following three literals (corresponding to the truth value assignment to three atoms):

$$[\![x \leqslant y]\!], \ \ [\![y \leqslant x]\!], \ \ \neg[\![x = y]\!] \tag{0}$$

This monome is then passed to the theories, where the theory of arithmetic detects an inconsistency and return the following lemma:

$$[\![x \leqslant y]\!] \wedge [\![y \leqslant x]\!] \ \Rightarrow \ [\![x = y]\!] \tag{1}$$

By conjoining this lemma to the original formula, the propositional assignment (0) is explicitly ruled out in the further reasoning performed by the SAT solver. Since (1) is a lemma, it could have been generated and conjoined to the input even before the first invocation of the SAT solver, but the strategy of generating this lemma on demand—that is, lazily—is the reason the architecture is called *lazy* proof explication.

```
Input: formula $F$
Output: satisfiability of $F$
while ($PSat$(F)) {
  let monome $m$ be the satisfying assignment ;
  $\mathcal{P} := CheckMonome(m)$ ;
  if ($\mathcal{P} = \emptyset$) {
    return $True$ ;
  } else {
    $F := F \wedge \mathcal{P}$ ;
  }
}
return $False$ ;
```

Fig. 0. Lazy-proof-explication algorithm without support for quantifiers.

Figure 0 outlines the algorithm of a theorem prover using lazy proof explication. $PSat(F)$ is implemented by calling an off-the-shelf SAT solver (after projecting the atoms onto propositional variables). If the result is $True$, a monome m is returned as the satisfying assignment. Then, $CheckMonome$ is called to determine if m is consistent with the underlying theories. $CheckMonome(m)$ returns a set of lemmas that are sufficient to refute monome m. An empty set indicates that the theories are unable to detect any inconsistency, in which case the algorithm reports that the original formula is satisfiable. Otherwise, the lemmas are conjoined to F and the loop continues until either the formula becomes propositionally unsatisfiable or the theories are unable to find inconsistency in the monome returned by $PSat$.

2 Handling quantifiers

When a formula contains quantifiers, usually the information expressed by the quantifiers must be used in showing a formula is unsatisfiable. This section discusses some basic notation and challenges for handling quantifiers. The main quantifier algorithm is presented in Section 3.

2.0 Terminology

A quantified formula has the form $(\,\delta\, x \bullet\ F\,)$, where δ is either $\forall$ or $\exists$. Quantifiers can be arbitrarily nested. Provided all the bound variables have been suitably α-renamed, the following three equations hold:

$$\begin{array}{rcl}
\neg(\,\delta\, x \bullet\ F\,) & \equiv & (\,\bar{\delta}\, x \bullet\ \neg F\,) \\
(\,\delta\, x \bullet\ F\,) \wedge G & \equiv & (\,\delta\, x \bullet\ F \wedge G\,) \\
(\,\delta\, x \bullet\ F\,) \vee G & \equiv & (\,\delta\, x \bullet\ F \vee G\,)
\end{array}$$

Here $\bar{\forall} = \exists$ and $\bar{\exists} = \forall$. By repeatedly applying the above three equations, we can move all the quantifiers in a quantifier atom to the front and convert it to the prenex form $(\,\delta_1\, x_1 \bullet\ (\,\delta_2\, x_2 \bullet\ \ldots\ (\,\delta_n\, x_n \bullet\ F\,)))$, where F does not contain any quantifier.

The existentially bound variables in the prenex form can be eliminated by skolemization. Skolemization replaces each existential variable x in the quantified body with a term $K(\Psi)$, where K is a fresh function symbol that is unique to the quantified formula and the existential variable x, and Ψ is the list of universally bound variables that appear before x. The skolem term $K(\Psi)$ is interpreted as the "existing term" determined by Ψ. We say the resulting purely universal formula is in *canonical form*. We use $Canon(Q)$ to denote the canonical form of a formula Q. Note that $Q \Rightarrow Canon(Q)$ is a lemma.

For any quantified formula C in canonical form and any substitution θ that maps each universal variable to a ground term, we write $C[\theta]$ to denote the formula obtained by taking C's body and applying θ to it.

2.1 Challenges in handling quantifiers

In order to reason about quantifiers, one can instantiate the universal variables with some concrete terms. This will introduce new facts that contain boolean structures, which cannot be directly used in the theory reasoning to refute the current monome. Neither can one only rely on propositional reasoning to handle these new facts because some inconsistency has to be determined by the theories. This means that in order to reason about quantifiers, both propositional reasoning and theory reasoning are necessary. This poses a challenge to theorem provers with lazy proof explication, where the two are clearly separated.

One approach is to conjoin the original formula with lemmas of instantiating universal quantifiers. Let Q be a quantifier literal in a formula F and let θ be a substitution that maps each universal variable in $Canon(Q)$ to a concrete term. Then, the following is a lemma for F:

$$Q \Rightarrow Canon(Q)[\theta]$$

It is a lemma because $Q \Rightarrow Canon(Q)$ and $Canon(Q) \Rightarrow Canon(Q)[\theta]$ are lemmas.

Conjoining these lemmas puts more constraints on the original formula. If the instantiations are properly chosen, more inconsistencies can be detected and eventually the formula can be shown to be unsatisfiable. Simplify [4] uses a matching heuristic to return a set of instantiations that will likely be useful in refuting the formula. However, there may still be too many useless instantiations returned by the matcher. This may blow up the SAT solver, because those lemmas can have arbitrary propositional structure, causing more case splits.

The quantifier algorithm we present in this paper adopts a different approach. First, the matching heuristic in Simplify is still used to return likely-useful instantiations. Then, the little SAT solver performs the propositional reasoning for those instantiated formulas. During the reasoning process, many new instantiations are generated, but only some of them are relevant in refuting the monome. Once the monome is refuted, our algorithm returns an appropriate lemma. The rationale of using the little SAT solver is to separate the propositional reasoning for finding a satisfying monome from the propositional reasoning for refuting a monome. Once a monome is refuted, many of the instantiations are not useful anymore. Without this two-tier approach, they would

```
Input: monome $m$
Output: a set of lemmas $\mathcal{P}$
procedure $CheckMonome(m) \equiv$
  Assert $m$ to the theories ;
  if $(m$ is inconsistent with the theories$)$ $\{$
    Theories output a lemma $\mathcal{P}$ that refutes $m$ ;
  $\}$ else $\{$
    Quantifier module generates lemmas $\mathcal{P}$ ;
  $\}$
  return $\mathcal{P}$ ;
```

Fig. 1. $CheckMonome$ algorithm in the one-tier technique.

remain in the formula and introduce many unnecessary case splits in the future rounds of reasoning.

3 Quantifier algorithm

The quantifier reasoning is performed in the $CheckMonome$ function in the algorithm shown in Figure 0. We show the quantifier algorithm in two steps. In section 3.0, we present the simple one mentioned in Section 2.1. In Section 3.1, we show how to use the little SAT solver in $CheckMonome$ to perform both propositional and theory reasoning.

3.0 One-tier quantifier algorithm

Figure 1 shows the $CheckMonome$ algorithm with the simple quantifier support discussed in Section 2.1. We call this the *one-tier* quantifier algorithm. The quantifier module is invoked only when the other theories cannot detect any inconsistency in the given monome. As discussed in Section 2.1, the lemmas are generated by instantiating universal quantifications. The instantiations are returned from a matching algorithm similar to that of Simplify. To avoid generating duplicate instantiations, the quantifier module remembers the instantiations it has produced. When no more lemmas can be generated, $CheckMonome$ will return an empty set, in which case the algorithm in Figure 0 will terminate with the result of $True$. To guarantee termination (at the cost of completeness), an implementation limits the number of times the quantifier module can be called for each run of the theorem prover and simply return an empty set when the limit is exceeded.

Unlike the lemmas output by the theories after discovering an inconsistency, the lemmas generated by the quantifier module generally are not guaranteed to refute the monome. There are two reasons for this. First, many inconsistencies involve both quantifier reasoning and theory reasoning. Without cooperating with the other theories, the lemmas returned by instantiating quantifiers alone may not be sufficient to propositionally block the monome. Second, the instantiations returned by the matching algorithm depend on the monome. Since instantiating quantifiers may produce more atoms to appear in a monome, it is possible that the matcher can provide the “right” instantiation only after several rounds.

As a result of $\mathit{CheckMonome}$ returning a set of lemmas insufficient to refute the monome, the next round may call $\mathit{CheckMonome}$ with the same monome, plus some extra literals coming from the quantifier instantiation. This is undesirable, because the SAT solver repeats its work to find the same monome again. A more serious problem of this simple algorithm is that many of the returned lemmas are not even relevant in refuting the monome. Those useless lemmas remain in the formula during the proving process, and without removing them, the SAT solver will eventually be overwhelmed by many unnecessary case splits.

3.1 Two-tier quantifier algorithm

In order to use quantifier instantiations to refute a monome, propositional reasoning is needed. The key problem of the simple $\mathit{CheckMonome}$ algorithm is: by directly returning the lemmas generated from quantifiers, it essentially relies on the main SAT solver to perform the propositional reasoning of those newly generated formulas. This causes repetitive and unnecessary work in the main SAT solver. To address this problem, we separate the propositional reasoning of the instantiated formula from that of the original formula by using a little SAT solver in our $\mathit{CheckMonome}$ algorithm (Figure 2).

Set $\mathcal{P}$ records all the lemmas generated so far. Some of the lemmas are generated by the quantifier module (line 8), while the others are generated by the theories (line 6). When new lemmas are added into $\mathcal{P}$, the little SAT solver performs propositional reasoning on $m \wedge \mathcal{P}$ (line 14). For every satisfying assignment $m \wedge m'$ produced by the little SAT solver, the algorithm invokes the theories to check if the assignment is consistent. To avoid redundant work, the algorithm checkpoints the state of the theories before entering the loop (line 2). As a result, the algorithm only needs to assert m' to the theories in each iteration (line 19).

The loop continues as long as $\mathcal{P}$ does not propositionally refute m and new lemmas are still being generated. If no more lemmas can be generated, either by the quantifier module or by the theories, and $m \wedge \mathcal{P}$ is still satisfiable, the algorithm terminates and returns an empty set, indicating failure to refute monome m.

Once $\mathcal{P}$ can refute m, the function $\mathit{FindUnsatCore}(m, \mathcal{P})$ is called to extract a good-quality lemma from $\mathcal{P}$. $\mathit{FindUnsatCore}(F, G)$ requires that $F \wedge G$ is propositionally unsatisfiable. It returns a formula H such that $G \Rightarrow H$, $F \wedge H$ is still unsatisfiable, and the atoms in H all occur in F.

Function $\mathit{FindUnsatCore}$ can be implemented in various ways. Modern SAT solvers can extract a small unsatisfiable core of a propositional formula [13] and this seems to be useful in $\mathit{FindUnsatCore}$. Alternatively, interpolants [9] may be used here, because any interpolant of G and F satisfies the specification of $\mathit{FindUnsatCore}(F, G)$. For our preliminary experiments, we have the following naive implementation of the $\mathit{FindUnsatCore}$ function for the particular kind of arguments that show up in the algorithm:

For a monome m and a formula P, if $\mathit{PSat}(m \wedge P) = \mathit{False}$, then there exists a minimal subset m_0 of m such that $\mathit{PSat}(m_0 \wedge P) = \mathit{False}$. Such a m_0 can be

Input: monome m
Output: a set of lemmas that can refute m, or $\emptyset$ when m is satisfiable

```
 0. procedure $CheckMonome(m) \equiv$
 1.   Assert $m$ to theories ;
 2.   Checkpoint all theories ;
 3.   $\mathcal{P} := \emptyset$ ;
 4.   loop {
 5.     if (the theories report inconsistency) {
 6.       Theories output a lemma $\mathcal{P}_0$ that refutes current monome ;
 7.     } else {
 8.       Quantifier module generates new lemmas $\mathcal{P}_0$ ;
 9.       if ($\mathcal{P}_0 = \emptyset$) {
10.         return $\emptyset$ ;
11.       }
12.     }
13.     $\mathcal{P} := \mathcal{P} \cup \mathcal{P}_0$ ;
14.     if ($PSat(m \wedge \mathcal{P}) = False$) {
15.       return $FindUnsatCore(m, \mathcal{P})$ ;
16.     }
17.     let $m \wedge m'$ be the satisfying monome ;
18.     Restore checkpoints in all the theories ;
19.     Assert $m'$ to theories ;
20.   }
```

Fig. 2. The $CheckMonome$ algorithm using the two-tier technique.

obtained by trying to take out one literal from m at a time and discard the literal if the formula remains propositionally unsatisfiable. It is easy to see that $P \Rightarrow \neg m_0$. We just return $\neg m_0$ as the result of $FindUnsatCore(m, P)$.

Correctness The correctness of the two-tier algorithm hinges on the fact that every formula in $\mathcal{P}$ is a lemma for the monome m. The correctness of the algorithm is formalized as the following theorem.

Theorem 1 *Let $\mathcal{P}$ be the set of all lemmas generated during the run of the algorithm. Then, the algorithm refutes the monome m iff $TSat(m \wedge \mathcal{P}) = False$.*

Intuitively, the result of the algorithm is the same as if we had generated all the lemmas $\mathcal{P}$ up front and run a standard Nelson-Oppen theorem prover on the formula $m \wedge \mathcal{P}$. Since conjoining lemmas does not change the satisfiability of a formula, the theorem shows our algorithm to be sound:

Corollary 2 *If the algorithm refutes m, then $QSat(m) = False$.*

This is because $\neg TSat(m \wedge \mathcal{P}) \Rightarrow \neg QSat(m \wedge \mathcal{P})$ by Proposition 0, and $QSat(m \wedge \mathcal{P}) = QSat(m)$ by the definition of lemma. On the other hand, the algorithm is not complete, since we cannot always generate all the lemmas relevant to the formula.

When the algorithm fails to refute a monome, all that is known is $TSat(m \wedge \mathcal{P}) = True$, that is, even with all the information in $\mathcal{P}$, a Nelson-Oppen theorem prover cannot refute the monome either.

4 Example

In this section, we demonstrate how our algorithm works on a small example.

Let P and Q be two quantified formulas:

$$\begin{array}{ll} P: & (\,\forall\, x \bullet\ x < 10 \Rightarrow R(f(x))\,) \\ Q: & (\,\forall\, y \bullet\ R(f(y)) \Rightarrow S(g(y))\,) \end{array}$$

where the match patterns to be used for P and Q are $x\colon f(x)$ and $y\colon g(y)$, respectively. A pattern ρ is a lambda term such that if a subterm t of the formula matches it, i.e. $\exists t_0 . t = \rho(t_0)$, t_0 will be used to instantiate the universal variable ρ is associated with. In our current implementation, the patterns are specified by the user, although they could be automatically inferred in most cases. We now trace our algorithm through the request of determining whether or not the following formula is satisfiable:

$$[\![P]\!] \ \wedge\ [\![Q]\!] \ \wedge\ ([\![b = 1]\!] \vee [\![b = 2]\!]) \ \wedge\ \neg[\![S(f(b))]\!] \ \wedge\ \neg[\![S(g(0)]\!])$$

In the first round, the main SAT solver returns a monome m, say

$$\{\ [\![P]\!],\ [\![Q]\!],\ [\![b = 1]\!],\ \neg[\![S(f(b)]\!]),\ \neg[\![S(g(0)]\!])\ \}$$

Since no theory can detect inconsistency, the quantifier module is invoked to generate lemmas. According to the match pattern, x is instantiated with b in P and y is instantiated with 0 in Q:

$$[\![P]\!] \ \Rightarrow\ ([\![b < 10]\!] \Rightarrow [\![R(f(b))]\!]) \tag{2}$$
$$[\![Q]\!] \ \Rightarrow\ ([\![R(f(0))]\!] \Rightarrow [\![S(g(0))]\!]) \tag{3}$$

The lemmas (2) and (3) are conjoined to the monome and the little SAT solver is called. The extended monome m' for the newly-introduced atoms might be:

$$\{\ \neg[\![b < 10]\!],\ \neg[\![R(f(0))]\!]\ \}$$

At this point the theories detect an inconsistency between $[\![b = 1]\!]$ and $\neg[\![b < 10]\!]$. So a new lemma is added:

$$[\![b = 1]\!] \wedge \neg[\![b < 10]\!] \ \Rightarrow\ False \tag{4}$$

In the next iteration, m' is

$$\{\ [\![R(f(b))]\!],\ \neg[\![R(f(0))]\!]\ \}$$

The theories are unable to detect inconsistency in the monome $m \wedge m'$. The quantifier module is invoked again to generate lemmas. This time the term $f(0)$ in the newly generated formulas matches the pattern, so x in P is instantiated by 0.

$$[\![P]\!] \ \Rightarrow\ ([\![0 < 10]\!] \Rightarrow [\![R(f(0))]\!]) \tag{5}$$

formula	one-tier SAT solver case splits	two-tier main SAT case splits	little SAT case splits	$\mathit{FindUnsatCore}$ case splits
ex2	7	1	1	9
ex9	44	9	3	23
ex100	428	106	2	104
prog.1.1	116	0	116	51
prog.2.2	547	4	491	277
prog.3.4	1919	13	1505	919
prog.3.4.err	2000	11	1312	218

Fig. 3. Results from running some preliminary experiments, showing the number of SAT solver case splits performed by the one-tier and two-tier approaches on some small examples.

The next m' is

$$\{\ [\![R(f(b))]\!],\ \neg[\![R(f(0))]\!],\ \neg(0 < 10)\ \}$$

The theory then detects an inconsistency:

$$\neg(0 < 10) \ \ \Rightarrow \ \ \mathit{False} \tag{6}$$

After conjoining (6), the original monome m will be propositional refuted. The lemma constructed is

$$[\![P]\!] \wedge [\![Q]\!] \wedge \neg[\![S(g(0))]\!] \ \ \Rightarrow \ \ \mathit{False}$$

After conjoining this lemma to the original formula, it becomes propositionally unsatisfiable.

If we use the simple algorithm, lemma (2) would be conjoined to the input formula, even though it has nothing to do with the contradiction. In the subsequent solving, this unnecessary lemma could introduce a case split on $([\![b = 1]\!] \vee [\![b = 2]\!])$, if the SAT solver happens to assign $[\![b < 10]\!]$ False. The theories would have to consider both in order to block the truth value assignment $\neg[\![b < 10]\!]$. By separating the two SAT solvers, our algorithm only needs to consider one of them.

5 Experimental results

We have implemented the two-tier quantifier algorithm in a lazy-proof-explication theorem prover in development at Microsoft Research. For a comparison, we also implemented the one-tier algorithm. This section describes the results from our preliminary evaluation of the algorithm.

Figure 3 shows the number of SAT-solver case splits required for some small examples. In addition to showing the case splits by the main and little SAT solvers in the two-tier approach, we show the number of case splits performed by our implementation of $\mathit{FindUnsatCore}$. We used two sets of examples, explained next.

/* axioms about operations that read and write the heap */

$(\forall\, h, x, F, y, G, a \bullet\;\; x \neq y \lor F \neq G \;\Rightarrow$
$\qquad\qquad sel(upd(h, x, F, a), y, G) = sel(h, y, G)\ ) \land$

$(\forall\, h, x, F, a \bullet\;\; sel(upd(h, x, F, a), x, F) = a\ ) \land$

/* fields names are distinct (only two distinctions are needed for this example) */

$f \neq g \land g \neq alloc \land$

/* object invariants hold initially, where H is the name of the heap */

$(\forall\, o \bullet\;\; o \neq null \land is(o, T) \land sel(H, o, alloc) \;\Rightarrow\; sel(H, o, f) < 7\ ) \land$

/* encoding of the call, where K is the name of the heap in the post-state */

$(\forall\, o, F \bullet\;\; sel(H, o, F) = sel(K, o, F) \lor$
$\qquad (o = p \land F = g) \lor (o = p \land F = f) \lor$
$\qquad \neg sel(H, o, alloc)\ ) \land$

$(\forall\, o \bullet\;\; sel(H, o, alloc) \;\Rightarrow\; sel(K, o, alloc)\ ) \land$

$(\forall\, o \bullet\;\; \neg sel(H, o, alloc) \land sel(K, o, alloc) \;\Rightarrow\; sel(K, o, f) < 7\ ) \land$

$sel(K, p, f) = 3 \land$

/* the (negation of the) postcondition to be proved */

$\neg(\forall\, o \bullet\;\; o \neq null \land is(o, T) \land sel(K, o, alloc) \;\Rightarrow\; sel(K, o, f) < 7\ )$

Fig. 4. A formula showing a typical structure of verification conditions of methods in an object-oriented program.

The first set of formulas was designed to show how the two-tier approach can save case splits over the one-tier approach. Formula $ex2$ is the example from Section 4, and $ex9$ and $ex100$ are the same example but with 9 and 100 different disjuncts instead of 2. The number of case splits for these examples (Figure 3) confirm that the two-tier approach can indeed reduced the number of case splits.

The second set of formulas was constructed to look like (the negations of) typical verification conditions of method bodies in an object-oriented program (*cf.* [8, 0]): on entry to the method body, one gets to assume that an object invariant holds for all allocated objects; on return from the method, one needs to show that the object invariant holds for all allocated objects; in between, the method body contains control structure and calls to other methods. In our example, we used an object invariant that puts an integer constraint on a field f of objects. In our example, the method body to be verified makes calls of the form $p.M(x)$, where p is some object and x is an integer. The semantics of the calls come from the specification of the callee. We used a specification for M that says that $p.M(x)$ sets the field $p.f$ to x, arbitrarily assigns to the field $p.g$, and allocates an arbitrary number of objects and changes the fields of those objects in arbitrary ways that satisfy the object invariant.

Formula $prog.1.1$ is (the negation of) the verification condition for a method whose body simply calls $p.M(3)$. It is shown in Figure 4. Formulas $prog.2.2$ and $prog.3.4$ are similar, but correspond to method bodies containing 2 and 4 calls (with various parameters) and with if statements that give rise to 2 and 3 execution paths, respectively. Formula $prog.3.4.err$ corresponds to the same program as $prog.3.4$, but with an inserted program error; thus, $prog.3.4.err$ is the only one of our small formulas that is satisfiable.

Since $prog.1.1$ is a straight-line program, there are no case splits to be done by the main SAT solver, so the little SAT solver performs roughly the same work as the SAT solver in the one-tier approach. Verification conditions produced from method bodies with more than one possible control-flow path contain disjunctions at the top level of the formula. As soon as there are such disjunctions in our examples, the two-tier approach performs fewer case splits not just in the main SAT solver, but in the main and little SAT solvers combined.

When the two-tier approach refutes a monome produced by the main SAT solver, the lemma returned to the main SAT solver has been pruned to contain a minimal number of literals. This keeps the state of the main SAT solver small, but the pruning has a price. The pruning may be done directly by the SAT solver, but our implementation performs the pruning using the $FindUnsatCore$ function described above. Figure 3 shows that this function performs a rather large number of case splits. We do not yet know the actual cost of these case splits relative to everything else in our implementation.

6 Discussion

6.0 Detecting useful quantifier instantiations

The two-tier technique separates the propositional reasoning of the input formula from the propositional reasoning of the quantifier instantiations. By doing so, this technique prevents useless instantiations from blowing up the propositional search of the input formula. However, it is possible for some instantiations to be repeatedly useful in refuting many propositionally satisfying assignment of the input formula. In such cases, it could be advantageous to expose this instantiation to the main SAT solver.

As an example, consider the following input formula:

$$[\![(\forall x \bullet\ P(x) \Rightarrow x \leqslant a\ )]\!] \ \wedge\ [\![P(2)]\!] \ \wedge\ ([\![a = 0]\!] \vee [\![a = 1]\!])$$

Suppose the main SAT solver picks a satisfying assignment consisting of the first two conjuncts and the disjunct $[\![a = 0]\!]$. The following instantiation

$$[\![(\forall x \bullet\ P(x) \Rightarrow x \leqslant a\ )]\!] \ \Rightarrow\ [\![P(2)]\!] \ \Rightarrow\ [\![2 \leqslant a]\!]$$

is sufficient to refute the current satisfying assignment. Consequently, the two-tier technique returns the following lemma:

$$[\![(\forall x \bullet\ P(x) \Rightarrow x \leqslant a\ )]\!] \wedge [\![P(2)]\!] \ \Rightarrow\ \neg[\![a = 0]\!]$$

However, the instantiation above is also sufficient to refute the (only) other satisfying assignment of the input formula.

If it is possible to detect such reuse of instantiations, the algorithm can expose these instantiations to the main SAT solver. We are currently exploring different heuristics to identify such useful instantiations.

6.1 Handling non-convex theories

For efficiency, it is best if theories combined using Nelson-Oppen are *convex*. Informally, a convex theory will never infer a disjunction of equalities without inferring one of them. Thus the decision procedures only need to propagate single equalities. For non-convex theories, sometimes it is necessary for the decision procedure to propagate a disjunction of equalities. For example, the integer arithmetic theory can infer the following fact:

$$0 \leqslant x \wedge x \leqslant 3 \quad \Rightarrow \quad x = 0 \vee x = 1 \vee x = 2 \vee x = 3.$$

This fact should be added as a lemma in the proving process. Like the lemmas generated by quantifier instantiation, there is a risk that useless lemmas increase the work required of the propositional search. The same technique discussed in this paper is readily applied to those non-convex theories. In this sense, our algorithm in Figure 2 actually provides a unified approach to handle both quantifiers and non-convex theories—they can both be viewed as a theory that can generate lemmas of arbitrary forms.

7 Related work

Among decision-procedure based theorem provers, besides our work, Simplify [4], Verifun [6], and CVC Lite [1] all provide some degree of quantifier support.

Simplify's method of using triggering patterns to find instantiations [10, 4] has proved quite successful in practice. Once an instantiation is generated, it remains in the prover until the search backtracks from the quantifier atom. We implemented a similar triggering algorithm and used a second SAT solver to reason about the instantiated formulas so that useful instantiations can be identified.

Our handling of quantifiers is based on Verifun's early work [7]. Some attempts have been made in Verifun to identify useful lemmas from instantiations of quantifiers. However, it seems that it is an optimization that works only when the instantiations alone can propositionally refute the current monome. In most scenarios, we believe, the quantifier module needs to cooperate with other theories to find out the instantiations that are useful to refute the monome.

In CVC Lite, each term is given a type and the formula is type checked. Types give hints about which terms can be used to instantiate a universal variable. However, instantiating a variable with every term whose type matches may be unrealistic for large problems.

Apart from the decision-procedure based theorem provers that rely on heuristic instantiations of quantified formulas, many automated first-order theorem provers including the resolution-based theorem provers (such as Vampire [12]) and the superposition theorem provers (such as HaRVey [3]) can handle quantifiers.

8 Conclusion

In this paper, we have proposed a two-tier technique for handling quantifiers in a lazy-proof-explication theorem prover. The propositional reasoning of the original formula and that of the instantiated formulas are handled by two SAT solvers. The major purpose of this separation is to avoid unnecessary case splits caused by intertwining useless instantiations and the original formula. The $FindUnsatCore$ method can extract, from a set of lemmas generated during quantifier reasoning, a "good lemma" that is both relevant to the problem and sufficient to refute the given monome. We also use checkpointable theories to improve efficiency during the quantifier reasoning.

Acknowledgments We thank the referees for their many useful comments on a previous version of this paper.

References

0. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
1. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer, July 2004.
2. Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, May 2002.
3. David Déharbe and Silvio Ranise. Light-weight theorem proving for debugging and verifying units of code. In *Proc. of the International Conference on Software Engineering and Formal Methods (SEFM03)*, Camberra, Australia, September 2003.
4. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
5. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
6. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *15th Computer-Aided Verification conference (CAV)*, July 2003.
7. Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Labs, 2004.
8. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
9. Kenneth L. McMillan. Interpolation and SAT-based model checking. In *15th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science. Springer, 2003.
10. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Also available as Xerox PARC technical report CSL-81-10, 1981.
11. Greg Nelson and Derek C. Oppen. Simplification by coorperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
12. Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2):91–110, 2002.
13. Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, May 2003.