An Explicating Theorem Prover for Quantified Formulas

Cormac Flanagan[1*], Rajeev Joshi[2], James B. Saxe
Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto
HPL-2004-199
November 2, 2004*

theorem proving, quantifiers, SAT solving, explication

Recent developments in fast propositional satisfiability solvers and proof-generating decision procedures have inspired new variations on the traditional Nelson-Oppen style of theorem provers. In an earlier paper, we described the design and performance of our explicating theorem prover *Verifun* for quantifier-free formulas over the theories of equality, rational linear arithmetic, and arrays. In this paper, we extend our original Verifun architecture to support universal and existential quantifiers, which arise naturally in many verification domains, and we verify key correctness properties of our design.

* Internal Accession Date Only
[1]Computer Science Department, Univ of California at Santa Cruz, Santa Cruz, CA *The author's work was partly supported by the NSF under Grant CCR-03411797 and by faculty research funds granted by the University of California at Santa Cruz [2] Laboratory for Reliable Software, Jet Propulsion Laboratory, Pasadena, CA
Approved for External Publication

An Explicating Theorem Prover for Quantified Formulas

Cormac Flanagan[1] *, Rajeev Joshi[2], and James B. Saxe[3]

[1] Computer Science Department, Univ of California at Santa Cruz, Santa Cruz, CA
[2] Laboratory for Reliable Software, Jet Propulsion Laboratory, Pasadena, CA
[3] Hewlett-Packard Laboratories, Palo Alto, CA

Abstract. Recent developments in fast propositional satisfiability solvers and proof-generating decision procedures have inspired new variations on the traditional Nelson-Oppen style of theorem provers. In an earlier paper, we described the design and performance of our explicating theorem prover *Verifun* for quantifier-free formulas over the theories of equality, rational linear arithmetic, and arrays. In this paper, we extend our original Verifun architecture to support universal and existential quantifiers, which arise naturally in many verification domains, and we verify key correctness properties of our design.

1 Introduction

Verifying formulas with respect to a collection of underlying theories requires a combination of both propositional and theory-specific reasoning. In an earlier paper [12], we described the *Verifun* theorem prover, which uses a new style of interaction between propositional and theory-specific decision procedures. This architecture cleanly separates the propositional search from the decision procedures for the underlying theories, thus allowing us to leverage recent advances in propositional SAT solving [14, 18]. Similar ideas have also been proposed by Barrett, Dill and Stump [3] and by de Moura and Rueß [8]. Although efficient for quantifier-free formulas, none of these prior approaches support formulas with quantifiers.

In contrast, many interesting problems in verification reduce to checking validity of quantified formulas. For example, the Extended Static Checkers for Modula-3 [10], Java [13], and multithreaded Java [11] use quantified formulas to express important correctness properties; for instance, that a particular object invariant must hold for all objects of a particular class, or that all elements of an array must satisfy a given property. The Simplify theorem prover [9], which forms the computational core of these extended static checkers, uses the classical Nelson-Oppen method with heuristic instantiation to reason about the validity of such quantified formulas.

* The author's work was partly supported by the NSF under Grant CCR-03411797 and by faculty research funds granted by the University of California at Santa Cruz.

In this paper, we explore how to incorporate a heuristic instantiation strategy into a modern, proof-explicating Nelson-Oppen prover such as Verifun. Our goal is to build a theorem prover for quantified formulas that combines the performance and scalability benefits of explicating theorem provers with the expressiveness of provers such as Simplify. Such an extended theorem prover would be valuable for many uses in verification; for example, it would enable more robust and scalable extended static checkers.

For simplicity, we describe Verifun as a satisfiability checker, which is the dual of a validity checker or theorem prover. The presentation of our results proceeds as follows. The following section describes the terminology used in this paper, and also reviews the underlying components on which Verifun is based. Section 3 presents the original Verifun architecture for quantifier-free formulas. Section 4 extends that architecture to support quantified formulas and includes a correctness proof for this design. The design is illustrated by means of an example in section 5, and in section 6, we discuss issues related to efficient implementation of our design. Section 7 discusses related work, and we conclude with section 8.

2 Background

2.1 Terminology and Notation

We use terminology that is standard in the literature. A *term* is a variable or an application of a function to a sequence of terms. An *atomic formula* is a propositional variable or an application of a predicate symbol to a sequence of terms. A *literal* is either an atomic formula or its negation, and a *clause* is a disjunction of literals. A *monome* is a conjunction of literals in which no atomic formula is both affirmed and negated. We identify a monome m with the partial truth assignment that assigns *true* to atomic formulas that are conjuncts of m and assigns *false* to atomic formulas whose negations are conjuncts of m. A *formula* is an arbitrary boolean combination of atomic formulas using the operators $\wedge, \vee$ and $\neg$. (We initially consider only quantifier-free formulas, and defer discussion of quantified formulas to Section 4.) We use the notation $F(x \leftarrow y)$ to denote the capture-free substitution of y for free occurrences of x within the formula F.

The task of a satisfier is to decide whether an input formula, called a *query*, is satisfiable for a given set of underlying theories. The underlying theories may assume particular semantics for some predicate and function symbols, such as "$>$" and "$+$", while leaving others uninterpreted.

We use the following notation to denote validity and satisfiability. Given a set of theories $\mathcal{T}$, we write $[F]_{\mathcal{T}}$ (read "everywhere F (wrt $\mathcal{T}$)") to denote that F is valid with respect to the theories. Similarly, we write $\langle F \rangle_{\mathcal{T}}$ (read "somewhere F (wrt $\mathcal{T}$)") to denote that F is satisfiable with respect to the theories. We also use the notations $[F]_{\mathcal{B}}$ and $\langle F \rangle_{\mathcal{B}}$ to denote that F is propositionally valid and propositionally satisfiable, respectively. We assume that these operators satisfy

the following laws, for any formula F and any monome m:

$$\langle m \rangle_{\mathcal{B}} \tag{1}$$

$$\langle F \rangle_{\mathcal{T}} \Rightarrow \langle F \rangle_{\mathcal{B}} \tag{2}$$

$$[F]_{\mathcal{T}} \wedge \langle m \rangle_{\mathcal{T}} \Rightarrow \langle F \wedge m \rangle_{\mathcal{T}} \tag{3}$$

Axiom (1) states that every monome is propositionally satisfiable; this follows from the definition of monomes above. Axiom (2) states that any formula that is satisfiable with respect to a set of theories $\mathcal{T}$ is also propositionally satisfiable. Axiom (3) states that if F is a tautology, then the satisfiability of $F \wedge m$ follows from the satisfiability of m.

The Verifun prover works via a collaboration between two components: a propositional satisfiability (SAT) solver and a proof-generating decision procedure. We review these two components next.

2.2 The SAT Solver

The SAT solver component decides the propositional satisfiability of a given formula F, and, in the case where F is satisfiable, returns a satisfying truth assignment for F. This satisfying truth assignment can be naturally represented as a monome m that entails the formula F. Thus, we assume that the SAT solver (procedure *satisfyProp*) obeys the following specification:

function *satisfyProp*(*Formula* F) : *Monome*
Returns m where m is either a monome, or the value *UNSAT*, such that
$m = \mathit{UNSAT} \Rightarrow \neg\langle F \rangle_{\mathcal{B}}$
$m \neq \mathit{UNSAT} \Rightarrow [m \Rightarrow F]_{\mathcal{B}}$

In practice, most current SAT solver implementations [14, 20] require that we first convert F to conjunctive normal form and replace each unique atomic formula with a fresh propositional variable. For simplicity of presentation, however, we ignore these details in our development.

2.3 The Proof-Generating Decision Procedure

The procedure *satisfyTheories* decides whether a given monome m is satisfiable with respect to the underlying theories $\mathcal{T}$. In our implementation, the individual theories are combined in the traditional Nelson-Oppen style [17] using equality sharing. If the monome m is unsatisfiable, the procedure *satisfyTheories* returns a "proof" of the unsatisfiability of m. This proof is represented as a formula R that is a tautology with respect to the theories (that is, $[R]_{\mathcal{T}}$) and which suffices to refute m via purely propositional reasoning (that is, $[R \Rightarrow \neg m]_{\mathcal{B}}$). For example, given the (unsatisfiable) monome:

$$(i = 0) \wedge (j = i + 1) \wedge \neg(j = 1) \wedge (k = i)$$

a suitable inconsistency proof is the tautology:

$$(i = 0 \wedge j = i + 1) \Rightarrow j = 1$$

from which the unsatisfiability of the monome follows via propositional reasoning. The specification of procedure *satisfyTheories* is as follows:

function *satisfyTheories*(*Monome* m) : *Formula*
Returns R where R is either a formula, or the value SAT, such that
$R = SAT \Rightarrow \langle m \rangle_{\mathcal{T}}$
$R \neq SAT \Rightarrow [R]_{\mathcal{T}} \wedge [R \Rightarrow \neg m]_{\mathcal{B}}$

3 A Satisfier for Quantifier-Free Formulas

The original Verifun architecture for quantifier-free formulas is shown in Figure 1. The procedure *satisfy* takes as input a quantifier-free formula F, and either determines that F is unsatisfiable (that is, $\neg\langle F \rangle_{\mathcal{T}}$), or it returns a satisfying assignment for F. This satisfying assignment is represented as a monome that is consistent with the underlying theories (that is, $\langle m \rangle_{\mathcal{T}}$), and which entails the query by propositional inference alone (that is, $[m \Rightarrow F]_{\mathcal{B}}$).

The procedure *satisfy* reasons about the satisfiability of F using a combination of propositional reasoning, performed by the procedure *satisfyProp*, and theory-specific reasoning, performed by the procedure *satisfyTheories*. The theory-specific reasoning is recorded in the explicated formula E, which serves to explicate at the propositional level the reasoning performed by *satisfyTheories* in refuting earlier truth assignments for F. The optimized search and backtracking algorithms of *satisfyProp* can then leverage this explicated information to quickly refute many other possible truth assignments.

The *satisfy* implementation works by repeatedly trying to find a propositional satisfying assignment m for the conjunction $F \wedge E$ of the query F with the explicated formula E. If the conjunction is unsatisfiable, then so is F, since the explicated formula E is a tautology. If the satisfying assignment m that is discovered is consistent with the theories, then m is a valid truth assignment for F. If m is inconsistent with the theories, then the call to *satisfyTheories* explicates a formula R that is sufficient in refuting m through propositional reasoning alone. The formula R is then conjoined with E, ensuring that the truth assignment m is never reconsidered on a subsequent loop iteration.

Experimental results indicate that Verifun's use of explicated clauses and a fast SAT solver enables it to scale much better to large problems than earlier provers such as Simplify [9]. For instance, on several of the UCLID benchmarks [5], Verifun improves over Simplify's performance by more than two orders of magnitude [12].

```
function satisfy(Formula F) : Monome
Returns m where m is either a monome, or the value UNSAT, such that
    m = UNSAT ⇒ ¬⟨F⟩_T
    m ≠ UNSAT ⇒ ⟨m⟩_T ∧ [m ⇒ F]_B
{
    Formula E := true;
    while (true) {
        Monome m := satisfyProp(F ∧ E);
        if (m = UNSAT) {
            return UNSAT;
        } else {
            Formula R := satisfyTheories(m);
            if (R = SAT) {
                return m;
            } else {
                E := E ∧ R;
            }
        }
    }
}
```

Fig. 1. The Verifun Satisfier for Quantifier-Free Formulas

4 Deciding Satisfiability of Quantified Formulas

We now describe how to extend the Verifun architecture outlined above to reason about *quantified formulas*. The following subsection introduces a number of additional notations and concepts necessary for this discussion, subsection 4.2 provides a specification of the extended satisfier for quantified formulas, subsection 4.3 describes the corresponding implementation, and subsection 4.4 contains a correctness proof sketch.

4.1 Terminology and Background

We extend our terminology to include the application of quantifiers. The definition of terms, atomic formulas, literals and clauses are as described in section 2.1. However, we extend the definition of a *formula* so that it can be an atomic formula, a boolean combination of formulas using the operators $\wedge, \vee$ and $\neg$, or a *quantified formula*. A quantified formula is either a universally quantified formula of the form $\forall x.F$, where F is a formula, or a negated universally quantified formula. Following tradition, we sometimes use the shorthand $\exists\, x.F$ as an abbreviation for $\neg\forall x.\neg F$. In addition, a monome can now contain both literals and quantified formulas as defined above. Also, we assume for convenience that the SAT solver (procedure *satisfyProp*) accepts a formula (with quantifiers) and returns either a satisfying monome or the special value *UNSAT* as before. In

doing so, we assume that the SAT solver treats each outermost quantified formula (that is, a quantified formula that is not nested inside another quantified formula) as an uninterpreted literal. In addition, we assume that all occurrences of the same outermost quantified formula are treated as the same literal.

We write $[F]_{\mathcal{QT}}$ (respectively, $\langle F \rangle_{\mathcal{QT}}$) to denote the validity (respectively, satisfiability) of a formula F with respect to the theories $\mathcal{T}$ and the standard interpretation of the universal-quantification symbol. In contrast, the judgments $[F]_{\mathcal{T}}$ and $\langle F \rangle_{\mathcal{T}}$ (used in the specification of *satisfyTheories*) do not interpret quantified formulas, but instead treat each unique outermost quantified formula as a fresh propositional variable. The judgments $[F]_{\mathcal{QT}}$ and $[F]_{\mathcal{T}}$ (and also $\langle F \rangle_{\mathcal{QT}}$ and $\langle F \rangle_{\mathcal{T}}$) coincide on quantifier-free formulas. Furthermore, a formula G that is $\mathcal{QT}$-satisfiable is also $\mathcal{T}$-satisfiable. That is, for any quantifier-free formula F and for any formula G:

$$[F]_{\mathcal{T}} \Leftrightarrow [F]_{\mathcal{QT}} \tag{4}$$

$$\langle F \rangle_{\mathcal{T}} \Leftrightarrow \langle F \rangle_{\mathcal{QT}} \tag{5}$$

$$\langle G \rangle_{\mathcal{QT}} \Rightarrow \langle G \rangle_{\mathcal{T}} \tag{6}$$

Supporting existential quantifiers (that is, universal quantifiers in negative positions) is straightforward, since we can replace each bound variable by a fresh (skolem) constant. For a quantified formula $\forall x.F$ occurring in a negative position, we write $\mathit{skolem}(x, F)$ to denote a fresh variable that is unique for the quantified formula $\forall x.F$. We say a monome m is *skolem-closed* if the information obtained by instantiating existentially-quantified formulas in this manner is already present in m, modulo propositional reasoning. More precisely, we define *skolem-closed*(m) to mean that for all formulas of the form $\neg\forall x.F$ in m,

$$[m \Rightarrow \neg F(x \leftarrow \mathit{skolem}(x, F))]_{\mathcal{B}}$$

Supporting universal quantifiers is more difficult. Verifun follows a strategy of *heuristic instantiation* of universally-quantified formulas. Although heuristic instantiation is incomplete, our experience with the Simplify theorem prover [9] has shown that heuristic instantiation is both efficient and sufficiently complete to be useful in practice [10, 13].

Since different instantiation heuristics may be appropriate in different situations, we abstract away the instantiation heuristics by assuming that for each universally-quantified formula $\forall x.F$ in a monome m, the function $\mathit{match}(m, x, F)$ describes the set of appropriate instantiations of that formula. In particular, $\mathit{match}(m, x, F)$ returns a set of substitutions for the bound variable x that determine which instantiations of $\forall x.F$ should be considered.

We define a monome m to be *match-closed* if the information obtained by performing all of these matching instantiations is already present in m, modulo propositional reasoning. More precisely, we define *match-closed*(m) to mean that for each quantified formula of the form $\forall x.F$ in m and for each substitution σ in $\mathit{match}(m, x, F)$,

$$[m \Rightarrow \sigma(F)]_{\mathcal{B}}$$

```
function satisfy(Formula F) : Monome
Returns m where m is either a monome, or the value UNSAT, such that
    m = UNSAT ⇒ ¬⟨F⟩_{QT}
    m ≠ UNSAT ⇒ ⟨m⟩_T ∧ [m ⇒ F]_B ∧ skolem-closed(m) ∧ match-closed(m)
{
    Formula E := true;
    while (true) {
        invariant Inv : [E]_{QT}
        Monome m := satisfyProp(F ∧ E);
        if (m = UNSAT) {
            assert q1 : ¬⟨F ∧ E⟩_B
L1 :        return UNSAT;
        } else {
            assert q2 : [m ⇒ F ∧ E]_B
            Formula R := checkMonome(m);
            if (R = SAT) {
                assert q3 : ⟨m⟩_T ∧ skolem-closed(m) ∧ match-closed(m)
L2 :            return m;
            } else {
                assert q4 : [R]_{QT} ∧ ¬[m ⇒ R]_B
L3 :            E := E ∧ R;
            }
        }
    }
}
```

Fig. 2. The Extended Verifun Satisfier for Quantified Formulas

As we will see, the notion of match-closed monomes allows us to write a specification of how complete Verifun is on universally-quantified formulas, and to prove that our implementation meets this specification.

4.2 Specification

The extended Verifun architecture for quantified formulas is described by the procedure *satisfy*, whose specification and implementation are shown in Figure 2. The procedure *satisfy* takes as input a formula F that may contain quantifiers. The procedure either determines that F is unsatisfiable with respect to the theories and the meaning of quantification (that is, $\neg\langle F\rangle_{\mathcal{QT}}$) and returns the special value *UNSAT*, or it returns a satisfying assignment for F. The satisfying assignment is a monome m that entails the query by propositional inference (that is, $[m \Rightarrow F]_{\mathcal{B}}$), and that is consistent with the underlying theories (that is, $\langle m\rangle_{\mathcal{T}}$). Note that, since our heuristic instantiation strategy is incomplete, m may not be satisfiable with respect to the meaning of quantifiers, and so there is no guarantee that $\langle m\rangle_{\mathcal{QT}}$. Instead, we specify a partial completeness property

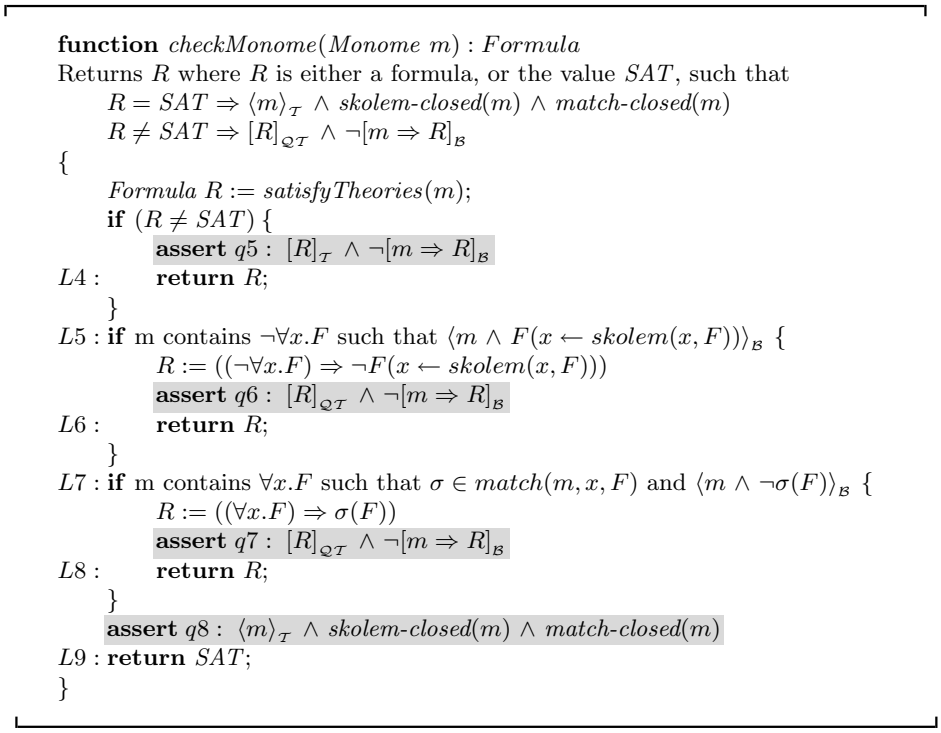

Fig. 3. The procedure *checkMonome*

for the procedure *satisfy* by stating that the satisfying assignment m must be both skolem-closed and match-closed.

4.3 Implementation

In the implementation of the procedure *satisfy* shown in Figure 2, the explicated formula E is a tautology that propositionally entails not only aspects of the semantics of the theories $\mathcal{T}$, but also aspects of the semantics of quantified formulas. The procedure *satisfy* repeatedly finds a monome m that propositionally implies the conjunction $F \wedge E$, and calls the procedure *checkMonome* on m. The procedure *checkMonome* (shown in Figure 3) attempts to explicate a formula refuting m using a sequence of strategies, as described below. First, *checkMonome* tries to prove that m is inconsistent with the theories by calling *satisfyTheories*. If *satisfyTheories* explicates a proof R that refutes m, then *checkMonome* returns that proof, and *satisfy* conjoins R to E. Since $[R \Rightarrow \neg m]_{\mathcal{B}}$, the conjoined formula $E \wedge R$ ensures that the SAT solver will never return the same truth assignment m on a subsequent loop iteration.

If m is consistent with the theories, the procedure *checkMonome* checks whether m is skolem-closed by searching for a formula $(\neg\forall x.F)$ in m such that the instantiation $\neg F(x \leftarrow skolem(x, F))$ is not propositionally implied by m,

that is:

$$\neg[m \Rightarrow \neg F(x \leftarrow skolem(x, F))]_{\mathcal{B}}$$

This reduces to the propositional SAT query $\langle m \wedge F(x \leftarrow skolem(x, F))\rangle_{\mathcal{B}}$, which is easily discharged using the SAT solver. (We discuss alternative implementation techniques in Section 6.)

If an existentially quantified formula is found that satisfies these conditions, then *checkMonome* explicates and returns the following instantiation R:

$$(\neg\forall x.F) \Rightarrow \neg F(x \leftarrow skolem(x, F))$$

Clearly, R is a tautology (that is, $[R]_{\mathcal{QT}}$). In addition, R explicates information that is not present in the monome m, and so m is not a truth assignment for R, that is, $\neg[m \Rightarrow R]_{\mathcal{B}}$. On subsequent iterations, the SAT solver can leverage this explicated rule to avoid repeatedly producing the same truth assignment m.

Finally, *checkMonome* checks whether m contains a universally-quantified formula $\forall x.F$ that has a matching substitution $\sigma \in match(m, x, F)$ such that the instantiation $\sigma(F)$ is not propositionally implied by m, that is, $\neg[m \Rightarrow \sigma(F)]_{\mathcal{B}}$. Again, the last check reduces to the SAT query $\langle m \wedge \neg\sigma(F)\rangle_{\mathcal{B}}$. If such a formula is found, *checkMonome* explicates the following instantiation rule, which communicates part of the meaning of universal quantifiers to the SAT solver, ensuring that the truth assignment m is never reconsidered on a subsequent iteration.

If none of the conditions above hold, the monome m is both skolem-closed and match-closed, and is satisfiable with respect to the theories. The procedure *checkMonome* reports that m is satisfiable, and the procedure *satisfy* then returns m as a satisfying assignment for the original query F.

4.4 Correctness

To show that *satisfy* implements its specification[4], we first observe that correctness of assertions $q1$ and $q2$ follows directly from postconditions of *satisfyProp*, while correctness of assertions $q3$ and $q4$ follows directly from postconditions of *checkMonome*. Next, we note that Inv is a loop invariant: it holds initially because E is initially *true*. To show that it is preserved by each iteration, we need to show that $[E \wedge R]_{\mathcal{QT}}$ holds before the assignment at label $L3$ is executed. But this follows directly from the first conjunct of $q4$, the inductive hypothesis $[E]_{\mathcal{QT}}$, and the distributivity of $\wedge$ over $[\cdot]_{\mathcal{QT}}$.

It remains to show that *satisfy*'s two postconditions hold at labels $L1$ and $L2$, respectively. The former holds because $q1 \wedge Inv$ holds at label $L1$, from which we can derive the postcondition $\neg\langle F\rangle_{\mathcal{T}}$ by using the following result:

$$\neg\langle F \wedge E\rangle_{\mathcal{B}} \wedge [E]_{\mathcal{T}} \;\Rightarrow\; \neg\langle F\rangle_{\mathcal{T}}$$

which follows from axioms 2 and 3 and boolean logic. From this, using axiom 6, the result follows. The remaining proof, that the postcondition

$$\langle m\rangle_{\mathcal{T}} \wedge [m \Rightarrow F]_{\mathcal{B}} \wedge \textit{skolem-closed}(m) \wedge \textit{match-closed}(m)$$

[4] Note, however, that the procedure *satisfy* may not terminate.

holds at label $L2$, follows directly from $q2$, $q3$ and boolean logic.

To demonstrate that the procedure *checkMonome* implements its specification, we first observe that the assertion $q5$ follows from the postcondition of *satisfyTheories*. We assume that *satisfyTheories* returns quantifier-free formulas, and hence by axiom 4 the property $[R]_{\mathcal{QT}}$ also holds at label $L4$, which ensures that the return statement at label $L4$ satisfies the postcondition:

$$[R]_{\mathcal{QT}} \wedge \neg[m \Rightarrow R]_{\mathcal{B}}$$

Next, the assumptions $q6$ and $q7$ follow by boolean logic, and the meaning of quantifiers, and hence the return statements at $L6$ and $L8$ also satisfy the postcondition. Finally, if none of the **then** branches in *checkMonome* are taken, assumption $q8$ follows from the postcondition of *satisfyTheories*, the definitions of skolem-closed and match-closed, and boolean logic, and hence the return statements at label $L9$ satisfy the postcondition:

$$\langle m \rangle_{\mathcal{T}} \wedge \textit{skolem-closed}(m) \wedge \textit{match-closed}(m)$$

5 Example

In this section, we illustrate our ideas by showing how our approach might work in determining the satisfiability of the following quantified formula F:

$$\begin{aligned} & b \geq 1 \\ \wedge\ & b > 0 \Rightarrow \forall x.(P(x) \vee \neg\forall y.Q(x,y)) \\ \wedge\ & \neg P(a) \\ \wedge\ & \forall z.Q(a,z) \end{aligned}$$

For this example, our prover might proceed as follows.

- Suppose the first call to *satisfyProp* on the formula F returns the monome:

  $$m_0 \equiv (b \geq 1) \wedge \neg(b > 0) \wedge \ldots$$

  This monome causes *checkMonome* to invoke *satisfyTheories*, which returns the explicated proof

  $$R_0 \equiv (b \geq 1 \Rightarrow b > 0)$$

- Next, *satisfyProp* is reinvoked on $(F \wedge R_0)$ and returns the monome

  $$m_1 \equiv (b \geq 1) \wedge (b > 0) \wedge (\forall x.(P(x) \vee \neg\forall y.Q(x,y))) \wedge \ldots$$

  On this monome, *satisfyTheories* returns SAT, and *checkMonome* executes the third **if** statement, labeled $L7$. Suppose that the matching routine, triggered by the presence of the atomic formula $P(a)$, returns the substitution $x := a$. This results in the instantiation $(P(a) \vee \neg\forall y.Q(a,y))$ of the quantifier body. Since the negation of this instantiation is not propositionally inconsistent with the monome, the procedure *checkMonome* explicates the following proof

  $$R_1 \equiv (\forall x.(P(x) \vee \neg\forall y.Q(x,y))) \Rightarrow (P(a) \vee \neg\forall y.Q(a,y))$$

– The next call to *satisfyProp* on $(F \wedge R_0 \wedge R_1)$ returns the monome

$$m_2 \equiv \neg\forall y.Q(a,y) \wedge m_1$$

Again, *satisfyTheories* returns SAT, and causes *checkMonome* to execute the second **if** statement, labeled $L5$. Let K denote the skolem variable *skolem*$(y,Q(a,y))$. Since the check $\langle m2 \wedge Q(a,K)\rangle_{\mathcal{B}}$ is successful, procedure *checkMonome* explicates a new proof

$$R_2 \equiv (\neg\forall y.Q(a,y)) \Rightarrow \neg Q(a,K)$$

– The next call to *satisfyProp* on $(F \wedge R_0 \wedge R_1 \wedge R_2)$ returns a monome

$$m_3 \equiv \neg Q(a,K) \wedge m_2$$

Again, *satisfyTheories* returns SAT. The condition in the second **if** statement now fails, since $m3$ contains $\neg Q(a,K)$. Suppose that the matching routine, triggered by the presence of $Q(a,K)$, now instantiates the last quantifier with the instantation $z := K$, causing the following proof to be explicated at line $L8$

$$R_3 \equiv (\forall z.Q(a,z)) \Rightarrow Q(a,K)$$

– Finally, the next call to *satisfyProp* on $(F \wedge R_0 \wedge R_1 \wedge R_2 \wedge R_3)$ returns $UNSAT$, indicating that the original formula was unsatisfiable.

6 Implementation Considerations

In this section, we discuss various implementation issues that are important in order to improve the performance of our approach.

First, as discussed in section 4.1, we have so far conveniently assumed that *match*(m,x,F) produces all relevant instantations for $\forall x.F$. In both Verifun and Simplify, this is achieved by associating with each universally-quantified formula an associated *pattern*, where a pattern is a set of terms that contains all the bound variables[5] of the quantifier. The function *match*(m,x,F) is then implemented by searching for all matches (upto equivalence) of this pattern in the *E-graph* data structure [17, 9], which represents all equivalences and congruences inferred from m. Patterns may be provided either explicitly, or inferred heuristically, by examining the syntactic structure of the quantifier body F, as done by Simplify [9]. In addition, the use of various performance optimizations (such as *fingerprinting* and the *mod-time* and *pattern-element* optimizations [9]) can yield considerable improvement in the performance of matching. A matching procedure which implements these heuristics and optimizations can be directly used in our approach.

[5] Though we have not discussed it in this paper, the generalization of our definitions to quantifiers with more than one bound variable is fairly straightforward.

Second, note that the guards to the **if** statements in lines $L5$ and $L7$ require checking whether certain formulas of the form $m \wedge P$ are propositionally satisfiable. While this may seem like an expensive check, in practice it is often cheaper, because modern SAT solvers often support the ability to solve such similar SAT problems incrementally. Thus, even though we invoke the SAT solver repeatedly to check the **if** condition for several quantifier bodies, by incrementally invoking the SAT solver from the state it reaches after asserting the literals in m, we can substantially reduce the performance cost of these calls.

A third point to note is that we can also safely weaken the guards in these two **if** statements. Since the formula R that is returned is always a tautology, such a weakening may cause the algorithm to explicate unnecessary proofs, but it does not introduce unsoundness. One such weakening, which we used in the first prototype implementation of quantifiers in Verifun, does not invoke the SAT solver. Instead, it maintains two sets E and A, where E contains the set of all quantified formulas that have been existentially instantiated (in line $L6$), and A contains a set of pairs of the form (quantified formula, substitution) recording each heuristic instantiation that was returned (in line $L8$). Using these sets, the guards in lines $L5$ and $L7$ may be weakened as follows:

$L5'$: **if** m contains $\neg\forall x.F$ such that $\neg\forall x.F \notin E$ {

$L7'$: **if** m contains $\forall x.F$ such that $\sigma \in match(m, x, F)$ and $(\forall x.F, \sigma) \notin A$ {

In addition, sets E and A are updated in the obvious way.

Finally, we note that even the design presented in section 4.3 has the potential limitation that it may explicate a great many instantiation rules for universally-quantified formulas. Even instantiation rules that have not proved valuable in refuting conjectured truth assignments are explicated to the SAT solver, and these many "useless" rules may seriously degrade the performance of the SAT solver.

To avoid this problem, we can refine the procedure *checkMonome* to only return useful instantiation rules for universally-quantified formulas. The refined implementation replaces the last **if** statement of Figure 3 with the following code:

```
FormulaSet S := ∅;
Monome m' := m;
while (m' contains ∀x.F such that σ ∈ match(m', x, F) and ⟨m' ∧ ¬σ(F)⟩_B) {
    invariant Inv : [m' ⇒ (m ∧ (∧S))]_B  ∧  (⟨m'⟩_B ⇔ ⟨m ∧ (∧S)⟩_B)
    S := S ∪ { (∀x.F) ⇒ σ(F) };
    m' := satisfyProp(m' ∧ σ(F));
    if (m' = UNSAT) {
        return getUnsatCore(m, S);
    }
}
```

Instead of returning any applicable instantiation rule, the new implementation records a set S of applicable instantiation rules, and attempts to extend

the monome m to a monome m' that is a satisfying assignment for $(m \wedge (\wedge S))$. We use the notation $(\wedge S)$ to denote the conjunction of all formulas in S. If the set S become sufficient to refute the current truth assignment m, then the procedure $getUnsatCore(m, S)$ is called, which has the specification:

function *getUnsatCore*(*Monome* m, *FormulaSet* S) : *FormulaSet*
Requires $\neg\langle m \wedge (\wedge S)\rangle_{\mathcal{B}}$
Returns a formula set $S' \subseteq S$ such that $\neg\langle m \wedge (\wedge S')\rangle_{\mathcal{B}}$

The procedure $getUnsatCore(m, S)$ computes and returns a small subset S' of S that is still sufficient to refute m. This procedure can be easily implemented by leveraging corresponding functionality in recent SAT solvers for extracting a small unsatisfiable core of a propositional formula [21]. Having thus determined a subset S' of instantiation rules that are actually useful in refuting m, this collection of useful instantiation rules is then returned as the result of *checkMonome*.

7 Related Work

The idea of theorem proving by solving a sequence of incrementally growing SAT problems occurs as early as the 1960 Davis and Putnam paper [7]. However, while their exhaustive enumeration algorithm is, in principle, complete for first-order predicate calculus, in practice it is likely to get overwhelmed by numerous irrelevant instantiations of quantified formulas.

The idea of extending decision procedures to generate proofs has been explored by George Necula in the context of his work on proof-carrying-code [15, 16]. However, our concerns are quite different: we are interested in proof-generation in order to produce a sufficient set of clauses to rule out satisfying assignments that are inconsistent with the underlying theory.

More similar in nature to our work is the Cooperating Validity Checker (CVC) [3], which also uses explicated clauses in order to control the boolean search. Recent work by de Moura and Ruess [8] also studies the impact of proof-explicating decision procedures in pruning the search space. Other systems employing a form of lazy explication include Math-SAT [1], which is specialized to the theory of linear arithmetic, and Zapato [2], which is based on the Verifun algorithm, but uses a decision procedure due to Harvey and Stuckey for solving a restricted class of arithmetic constraints. All of these systems focus on quantifier-free formulas. We build on this work by extending these ideas to also support quantified formulas via heuristic instantiation.

Verifun produces propositional projections of theory-specific facts lazily, on the basis of its actual use of those facts in refuting proposed satisfying assignments. An alternative approach is to eagerly generate all theory-specific facts that might possibly be needed for testing a particular query in a pre-processing phase. This eager approach has been applied by Bryant, German, and Velev [4] to the domain of equality with uninterpreted function symbols and arrays, and extended by Bryant, Lahiri and Seshia [6] to incorporate counter arithmetic. Ofer

Strichman [19] has investigated the eager projection to SAT for Presburger and linear arithmetic. For certain domains, the eager approach has achieved impressive performance. For richer theories and in particular for problems involving quantification, it is unclear whether it will be possible to generate all necessary theory-specific facts at the outset without also including excessively-many irrelevant facts that would swamp the SAT solver.

8 Conclusion

Previous research [12, 3, 8, 2] has helped make the case that leveraging the use of fast SAT solvers and proof-generating decision procedures is a promising direction for achieving increased scalability and performance in theorem provers. However, so far, this work has focused only on quantifier-free theories. In this paper, we have described how to extend the approach to include formulas with universal and existential quantifiers, including arbitrarily nested quantifiers. Such quantified formulas are common in program verification [17, 10, 13, 11]. We have sketched a correctness proof of our algorithm, shown how it can be made as complete as a prover such as Simplify [9], and discussed several key implementation issues.

References

1. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the 18th Conference on Automated Deduction*, July 2002.
2. Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proceedings 16th International Conference on Computer Aided Verification*, July 2004.
3. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, July 2002.
4. Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 470–482. Springer, July 1999.
5. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings. In *Proceedings of the First International Workshop on Constraints in Formal Verification*, September 2002.
6. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, July 2002.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.

8. Leonardo de Moura and Harald Ruess. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002.
9. David Detlefs, Greg Nelson, and James B. Saxe. A theorem-prover for program checking. Technical Report HPL-2003-148, HP Systems Research Center, July 2003.
10. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, Compaq Systems Research Center, 1998.
11. C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *CAV 02: Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 180–194, 2002.
12. Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem Proving using Lazy Proof Explication. In *Proceedings 15th International Conference on Computer Aided Verification*, July 2003.
13. 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)*, pages 234–245, 2002.
14. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, June 2001.
15. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Mellon University, 1998. Also available as CMU Computer Science Technical Report CMU-CS-98-154.
16. George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 25–44, June 2000.
17. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979. A revised version of this thesis was published as Xerox PARC Computer Science Laboratory Research Report CSL-81-10.
18. João Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositionsal satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
19. Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In *Proceedings Formal Methods in Computer-Aided Design, 4th International Conference*, pages 160–170, 2002.
20. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, November 2001.
21. Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.