Quantified Invariant Generation Using an Interpolating Saturation Prover

K.L. McMillan

Cadence Berkeley Labs

Abstract. Interpolating provers have a variety of applications in verification, including invariant generation and abstraction refinement. Here, we extended these methods to produce universally quantified interpolants and invariants, allowing the verification of programs manipulating arrays and heap data structures. We show how a paramodulation-based saturation prover, such as SPASS, can be modified in a simple way to produce a first-order interpolating prover that is complete for universally quantified interpolants. Using a partial axiomatization of the theory of arrays with transitive closure, we show that the method can verify properties of simple programs manipulating arrays and linked lists.

1 Introduction

An interpolating prover derives an interpolant for a pair (or in general a sequence) of logical formulas from a proof of unsatisfiability of those formulas. An interpolant for a pair of formulas (A, B) is a formula over their common vocabulary that is implied by A and inconsistent with B. Interpolating provers have been used to generate inductive invariants for proving properties of sequential circuits [7] and sequential programs [9], as well as abstraction refinement [4]. However, their use so far has been limited to propositional logic (with a Boolean satisfiability solver) or quantifier-free first-order logic for fixed theories (with a ground decision procedure) [8]. While effective, these methods are strongly limited in their ability to handle programs manipulating arrays and heap data structures because these generally require quantified invariants.

In this paper, we show how to modify a paramodulation-based prover for first order logic (FOL) with equality to produce an interpolating prover. This prover is complete for generation of universally quantified interpolants (though the input formulas may be in full FOL). Because it is a full first order prover, it allows us to introduce various theories that may be useful for expressing invariants by axiomatizing them. For example, we show that an incomplete axiomatization of FO(TC), the first-order theory of transitive closure, allows us to verify properties of simple heap-manipulating programs.

The primary problem that we must solve in making a practical interpolating prover is *divergence* of the interpolants. That is, we generate inductive invariants from the interpolants obtained by refuting unwindings of the program of increasing length. If these interpolants diverge with increasing unwinding length (for

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 413–427, 2008.

example by exhibiting increasing numeric constants or function nesting depth or number of quantifiers) then this approach fails. This problem was solved for quantifier-free case (for certain theories) in [5]. Here, we solve the problem in a different way, by bounding the clause language of the saturation prover. We show that the method is complete for universally quantified invariant generation, that is, if there is an inductive invariant proving a given property, we are guaranteed to find one eventually.

We also show experimentally, by modifying the SPASS prover [14] that the method does in fact converge for some simple example programs manipulating arrays and linked lists.

Related work. Indexed predicate abstraction [6] is a method that can generate the strongest universally quantified inductive invariant of a program over a fixed set of atomic predicates. However, some of these atomic predicates typically must be provided manually, as effective selection algorithms are lacking. Moreover, the forward image operator in this method is problematic, requiring in the worst case an exponential number of calls to a decision oracle for first-order logic. The method presented here does not require an image operator or a decision oracle. It may, however, provide a useful heuristic for indexed predicate refinement.

Since the method presented here can handle FO(TC), it is comparable in power to canonical heap abstraction [12]. The abstract states in this method (with reachability predicates) can be expressed as formulas in FO(TC). The difference between the methods is therefore mainly a matter of efficiency, which remains to be determined. However, the interpolation method has the advantage that it does not require the user to provide instrumentation predicates manually. It could be that interpolation in FO(TC) will be a useful approach for automated refinement in the canonical abstraction method.

Finally, the method can also be compared to parameterized invariant generation methods such as [13]. The main advantage of interpolation is that it can synthesize the Boolean structure of the invariant, and it can handle heap properties using transitive closure that cannot be handled by parameterized methods. On the other hand, the arithmetic reasoning ability of the present approach is limited compared to these methods.

2 Background: Paramodulation Calculus

Paramodulation [11] is the method of choice for proving first order formulas with equality. We begin by describing the basic principles of saturation provers based on paramodulation. This is necessarily a quick review. The material in this section is derived from an excellent article by Niewenhuis and Rubio [10], to which the reader is referred for greater depth.

Preliminaries. Let Σ be a countable vocabulary of function and predicate symbols, with associated arities. Function symbols with arity zero will be called constants. We assume that Σ contains at least one constant. We will use meta-variables f, g, h to represent function symbols, a, b, c to represent constants, and

P, Q to represent predicate symbols. We will also distinguish a finite subset Σ_I of Σ as *interpreted* symbols. In particular, we assume that Σ_I contains the binary predicate symbol $\simeq$, representing equality. Let $\mathcal{V}$ be a countable set of variables, distinct from Σ. We will use U, V to represent variables. The set of terms $\mathcal{T}$ is the least set such that $\mathcal{V} \subseteq \mathcal{T}$ and for every function symbol f of arity k, and terms $t_1 \ldots t_k \in \mathcal{T}^k$, we have $f(t_1, \ldots, t_k) \in \mathcal{T}$. We will use s, t (and sometimes l, r) to represent terms. The vocabulary of a term or formula ϕ, denoted $L(\phi)$ is the set of uninterpreted symbols occurring in ϕ. If S is a vocabulary, we let $\mathcal{T}(S)$ denote the set of terms t such that $L(t) \subseteq S$. Similarly, $\mathcal{L}(S)$ is the set of first-order formulas ϕ such that $L(\phi) \subseteq S$. We will also write $\mathcal{L}(\phi)$ for $\mathcal{L}(L(\phi))$.

An *atom* is $P(t_1, \ldots, t_k)$, where P is a k-ary predicate symbol and $t_1, \ldots, t_k$ are terms. A *literal* is an atom or its negation. A *clause* is a disjunction of literals in which the variables are implicitly universally quantified. Following tradition, we will write clauses in the form $\Gamma \rightarrow \Delta$, where Γ is the multiset of negative literals in the clause, and Δ is the multiset of positive literals. Also following tradition, we will write a formula multiset as list of formulas and formula multisets. Thus, if Γ is a multiset of formulas and ϕ a formula, then Γ, ϕ represents $\Gamma \cup \{\phi\}$.

A substitution σ is a map from variables to terms. For any term of formula ϕ, we write $\phi\sigma$ to indicate the simultaneous substitution in ϕ of $\sigma(U)$ for all free occurrences of U, for all variables U in the domain of σ. A formula or term is said to be *ground* if it contains no variables. A substitution is ground if all terms in its range are ground. The *ground instances* of a clause C are all the clauses $C\sigma$, where σ is a ground substitution over the variables in C. A *position* p is a finite sequence of natural numbers, representing a syntactic position in a term or formula. If ϕ is a formula or term, then $\phi|_p$ represents the subformula or subterm of ϕ at position p. Thus, $\phi|_\epsilon$ is ϕ itself, $\phi|_i$ is the i-th argument of ϕ, $\phi|_{ij}$ is the j-th argument of the i-th argument, and so on. The notation $\phi[\psi]_p$ means ϕ with ψ substituted in position p.

Paramodulation with constrained clauses. Paramodulation provers use the concept of a *reduction order* to reduce that amount of deduction that is required for completeness. For our purposes, a reduction order $\succ$ is a total, well-founded order on ground terms that is *monotonic* and has the *subterm property.* Monotonicity means that whenever $\psi_1 \succ \psi_2$, we have $\phi[\psi_1]_p \succ \phi[\psi_2]_p$. The subterm property says that $\phi \succ \phi|_p$ for all $p \neq \epsilon$. A reduction order can be extended to finite multisets of formulas. Given two multisets S and S', we say $S \succ S'$ if $S(\phi) > S'(\phi)$, where ϕ is the maximal formula such that $S(\phi) \neq S'(\phi)$. This allows us to totally order the ground clauses with respect to $\succ$.

We will be concerned here with refutation systems that take a set of clauses, and try to prove that the set is unsatisfiable by deriving the empty clause (equivalent to false). For purposes of refutation, a clause with variables is logically equivalent to the set of its ground instances (this is a consequence of Herbrand's theorem). Thus, it is useful to think of a clause with variables as simply a pattern abbreviating a countable set of ground clauses. To describe the operation of a paramodulation prover, it is useful to introduce the notion of a *constrained*

clause. This is written in form $C \mid T$ where C is a clause, and T is a constraint. The constraint is usually a conjunction of constraints of the form $s = t$ or $s > t$, where s and t are terms or atoms. For a given ground substitution σ, $s = t$ means $s\sigma$ and $t\sigma$ are the syntactically equal, and $s > t$ means $s\sigma \succ t\sigma$. The interpretation of $C \mid T$ is the set of all ground instances $C\sigma$ of C such that $T\sigma$ is true. For example, $P(U, V) \mid U > a$ means that P holds of all pairs of ground terms U, V, such that $U \succ a$. Note that a clause with an unsatisfiable constraint is by definition equivalent to true and an empty clause with a satisfiable constraint is equivalent to false.

An *inference* is the derivation of a constrained clause (the *conclusion*) from a multiset of constrained clauses (the *premises*) and is written in this form:

$$\frac{C_1 \mid T_1 \ldots C_n \mid T_n}{D \mid T}$$

An *inference rule* is a pattern that finitely describes a set of valid inferences. For example, here is the rule for resolution:

$$\frac{\Gamma \to \Delta, \phi \mid T_1 \quad \Gamma', \phi' \to \Delta' \mid T_2}{\Gamma, \Gamma' \to \Delta, \Delta' \mid \phi = \phi' \wedge T_1 \wedge T_2}$$

Note that because of the constraint $\phi = \phi'$ in the conclusion, every ground instance of this inference is valid. Most resolution provers eliminate the constraint $\phi = \phi'$ by substituting with σ, the most general unifier of ϕ and ϕ', yielding $(\Gamma, \Gamma' \to \Delta, \Delta')\sigma \mid (T_1 \wedge T_2)\sigma$. If ϕ and ϕ' cannot be unified, the conclusion's constraint is unsatisfiable, and the inference is discarded. In the sequel, we will omit the constraints on the premises and take it as implied that these constraints are inherited by the conclusion.

For refutation in the theory of equality, most modern provers use a superposition calculus (since resolution, though complete, is very inefficient for this purpose). This is based on substitution of equals for equals. Here is an example of a superposition inference:

$$\frac{P \to f(x) = y \quad Q \to x = z}{P, Q \to f(z) = y}$$

We say we have performed superposition *with* $x = z$, *into* $f(x) = y$. This approach can generate an enormous number of inferences. However, we can reduce this chaos by using *ordered superposition*. That is, we only need to perform the above inference if x and $f(x)$ are *maximal* terms in their respective clauses, with respect to $\succ$. Intuitively, we are always rewriting downward in the order. The inference rules for ordered superposition are as follows:

$$\text{superposition right: } \frac{\Gamma \to \Delta, s \simeq t \quad \Gamma' \to \Delta', l \simeq r}{\Gamma, \Gamma' \to \Delta, \Delta', s[r]_p \simeq t \mid s|_p = l \wedge OC}$$

$$\text{superposition left: } \frac{\Gamma, s \simeq t \to \Delta \quad \Gamma' \to \Delta', l \simeq r}{\Gamma, \Gamma', s[r]_p \simeq t \to \Delta, \Delta' \mid s|_p = l \wedge OC}$$

$$\text{equality resolution: } \frac{\Gamma, s \simeq t \to \Delta}{\Gamma \to \Delta \mid s = t \wedge OC}$$

$$\text{equality factoring: } \frac{\Gamma \to s \simeq t, s' \simeq t', \Delta}{\Gamma, t \simeq t' \to s = t', \Delta \mid s = s' \wedge OC}$$

The equality resolution rule enforces reflexivity of equality, while the equality factoring rule eliminates redundant equalities. In each rule, OC is an ordering constraint. These constraints are not necessary, but they reduce the number of possible inferences greatly without sacrificing completeness. From our point of view, the only thing we need to know about OC is that it implies in the superposition rules that s and l are maximal terms in their respective clauses. Details can be found in [10].

We will call this system of inference rules $\mathcal{I}_\succ$, where $\succ$ is the reduction ordering used in the ordering constraints. Given any unsatisfiable set of "well-constrained clauses", $\mathcal{I}_\succ$ can derive false. The notion of "well-constrained" is too technical to present here (see [10]). We note only that clauses with constraints of the form $a < U$ are well-constrained, which is all the we require for present purposes. To be more precise, we have:

Theorem 1 ([10]). *For any reduction order $\succ$, system $\mathcal{I}_\succ$ is complete for refutation of well-constrained clause sets with equality.*

Note that this system handles only equality predicates. However, we can in principle translate any other predicate symbol P into a function symbol, such that $P(x, y, \ldots)$ is equivalent to $P(x, y, \ldots) = t$, where t is a symbol representing "true". Thus in principle, equality predicates are sufficient. In practice, provers typically retain arbitrary predicate symbols and also implement resolution.

Redundancy and saturation. A saturation prover has inference rules that deduce new clauses, and also *reduction rules* that delete redundant clauses. The prover is said to reach *saturation* when any new inference from existing clauses can be deleted by the reduction rules.

Relative to a set S of derived clauses and a reduction order $\succ$, a clause C is said to be *redundant* when it is entailed by clauses in S that are less than C. A more general notion is redundancy of inferences. An inference I is said to be redundant when its conclusion is entailed by clauses in S less than the maximal clause in its premises (for all ground instances satisfying its constraint). Intuitively, by deleting a redundant inference, we "postpone" the deduction of its conclusion (or lesser clauses entailing it). However cannot postpone its derivation infinitely, since the reduction order is well-founded.

A saturation prover starts with a set of "usable" clauses U, and an empty set of "worked off" clauses W. At each iteration of its main loop, it removes a clause G from U, called the *given clause*. Reduction rules are applied to G (possibly adding derived clauses to U). If G is not deleted, all possible inferences using G and some set of clauses from W are then generated, and G is added to W. If U becomes empty, then W is saturated.

The main results about saturation provers that are of interest here are the following:

1. If the set of inference rules is complete, and if only redundant inferences are deleted, and if the selection of given clauses is fair, then the saturation prover is complete for refutation.
2. Moreover, if any clause C is entailed by the original clauses, then eventually it is entailed by clauses in W that are less than C in the reduction order.

To express rules for deleting redundant inferences, we will introduce a notation for *replacement rules*. These have the form $I \overset{S}{\rightarrow} J$, where I is an inference, S is a clause set and J is a set of (sound) inferences. The intuitive meaning of a replacement is that, if clauses S are proved, adding inferences J makes inference I redundant. As an example, if $Q \succ a \succ b \succ P$, the following is a valid replacement:

$$\frac{P \quad P \rightarrow Q(a)}{Q(a)} \quad \overset{a=b}{\rightarrow} \quad \frac{P \rightarrow Q(a) \quad a = b}{P \rightarrow Q(b)}$$

That is, since $P \rightarrow Q(a)$ is greater than P, $P \rightarrow Q(b)$ and $a = b$, and these imply $Q(a)$, we have a valid replacement. For each given clause G, the prover checks whether I, the inference that produced G, can be deleted and replaced by other inferences J, using a replacement rule. Since this adds only valid inferences and deletes only redundant ones, both soundness and completeness are preserved.

3 Interpolants from Superposition Proofs

Given a pair of formulas (A, B), such that $A \wedge B$ is inconsistent, an *interpolant* for (A, B) is a formula $\hat{A}$ with the following properties:

- A implies $\hat{A}$,
- $\hat{A} \wedge B$ is unsatisfiable, and
- $\hat{A} \subseteq \mathcal{L}(A) \cap \mathcal{L}(B)$.

The Craig interpolation lemma [3] states that an interpolant always exists for inconsistent formulas in FOL.

We now show how to use a saturation prover to generate universally quantified interpolants from arbitrary formulas in FOL. The approach is based on generating local proofs:

Definition 1. *An inference is* local *for a pair of formulas* (A, B) *when its premises and conclusions are either all in* $\mathcal{L}(A)$ *or all in* $\mathcal{L}(B)$*. A proof is local for* (A, B) *when all its inferences are local.*

From a *local* refutation proof of A, B, we can derive an interpolant for the pair in linear time [5]. This interpolant is a Boolean combination of the formulas in the proof.

Unfortunately, it is easily shown that the superposition calculus described above is not complete if we restrict it to local proofs. Consider the case where A consist of the clauses $Q(f(a))$ and $\neg Q(f(b))$, while B contains $f(V) = c$ and $b, c \in L(B)$. An interpolant for (A, B) is $f(a) \not\simeq f(b)$. However, no local superposition

inferences are possible for these clauses. To solve this problem, we show that by choosing the precedence order appropriately and adding a replacement rule, we can force all the inferences to be local without sacrificing completeness, so long as A and B have an interpolant in $\mathcal{L}_\forall$. This yields a complete procedure for generating universally quantified interpolants.

Definition 2. *A reduction order* $\succ$ *is* oriented *for a pair of formula sets* (A, B) *when, for all terms or formulas* ϕ_1, ϕ_2 *over* $L(A, B)$*, if* $\phi_1 \notin \mathcal{L}(B)$ *and* $\phi_2 \in \mathcal{L}(B)$*, then* $\phi_1 \succ \phi_2$*.*

Intuitively, as we descend the order, we eliminate the symbols that occur only in A. One way to construct an oriented reduction order is to use the standard RPOS (recursive path ordering with status), setting the precedence order so that the symbols in $L(A) \setminus L(B)$ precede all the symbols in $L(B)$.

Now let us consider again our example of of incompleteness of local superposition. In this example, although no local superposition inferences are possible, we can make the non-local inference $Q(f(a)), f(V) \simeq c \vdash Q(c)$. We can then make the following *replacement*:

$$\frac{Q(f(a)) \quad f(V) \simeq c}{Q(c)} \quad \rightarrow \quad \frac{Q(f(a))}{f(a) \simeq U \rightarrow Q(U) \mid f(a) > U}$$

where U is a fresh variable. This replacement is valid for the following reasons. First, the right-hand inference is sound (that is, if Q holds of $f(a)$, then Q holds of any U equal to $f(a)$). Second, the conclusion on the right, $f(a) \simeq U \rightarrow Q(U) \mid f(a) > U$, when resolved with $f(V) \simeq c$, gives us $Q(c)$. Thus, the conclusion on the left is implied by proved clauses. Moreover, those clauses are both less than $Q(f(a))$ in the reduction order, given the constraint $f(a) > U$. That is, the conclusion on the left is implied by derived clauses less than its maximal premise. This means that adding the right inference makes the left one redundant.

We can now continue to construct a fully local refutation for our example problem, using replacements of this type:

1. $\rightarrow Q(f(a))$	(hypothesis from A)
2. $\rightarrow f(V) \simeq c$	(hypothesis from B)
3. $f(a) \simeq U \rightarrow Q(U)$	(superposition in 1 with 2, with replacement)
4. $Q(f(b)) \rightarrow$	(hypothesis from A)
5. $f(b) \simeq U,\ Q(U) \rightarrow$	(superposition in 4 with 2, with replacement)
6. $f(a) \simeq U,\ f(b) \simeq U \rightarrow$	(resolution of 3 and 5)
7. $f(b) \simeq c \rightarrow$	(resolution of 6 and 2)
8. $\rightarrow$	(resolution of 7 and 2)

Notice that the replacement allowed us to postpone the superposition steps until only symbols from B remained. For this reason, we will refer to this type of replacement as "procrastination". The procrastination rule for deletion of superposition right inferences can be stated as follows:

$$\frac{\Gamma \rightarrow \Delta, s \simeq t \quad \Gamma' \rightarrow \Delta', l \simeq r}{\Gamma, \Gamma' \rightarrow \Delta, \Delta', s[r]_p \simeq t \mid s|_p = l \wedge OC} \overset{*}{\rightarrow} \frac{\Gamma \rightarrow \Delta, s \simeq t}{s|_p \simeq U, \Gamma \rightarrow \Delta, s[U]_p \simeq t \mid s|_p > U}$$

where OC is the ordering constraint of the superposition right rule. The asterisk is to indicate that this rule is to be applied when p is *not* ϵ, the top position. This means that l is a strict subterm of s, and thus $s > l$.

Now we argue that this rule is valid. Call the left inference L and the right inference R. It is easily verified that R is sound. Now let LA_1, LA_2, LS, RA, RS stand respectively for the premises and conclusion of the left and right inferences. Since OC implies that $l > r$, we have $RS, LA_2 \models LS$, which we can prove by resolution. Finally, we need to show that $LA_1 \succ LA_2$ and $LA_1 \succ RS$. The former is guaranteed by the asterisk (that is, since $s > l$, and l is a maximal term in LA_1, we have $LA_1 \succ LA_2$). The latter is guaranteed by the constraint $s|_p > U$, which implies $s > s|_p > U$, and, by monotonicity, $s > s[U]_p$. Thus, procrastination right is a valid replacement.

The rule replacing superposition left inferences is similar:

$$\frac{\Gamma, s \simeq t \to \Delta \quad \Gamma' \to \Delta', l \simeq r}{\Gamma, \Gamma', s[r]_p \simeq t \to \Delta, \Delta' \mid s|_p = l \wedge OC} \xrightarrow{*} \frac{\Gamma, s \simeq t \to \Delta}{s|_p \simeq U, \Gamma, s[U]_p \simeq t \to \Delta \mid s|_p > U}$$

The argument for validity is similar to that for procrastination right. Since the procrastination rules are valid replacements, meaning that they only generate sound inferences and delete redundant ones, we have immediately that:

Lemma 1. *System $\mathcal{I}_\succ$ with procrastination is complete for refutation for well-constrained clause sets.*

We now observe that, for pairs (A, B) with universally quantified interpolants, we require only local ground instances of B clauses for refutation completeness.

To be more precise, if C is a clause, let $C \mid L(B)$ stand for the set of ground instances $C\sigma$ of C where the range of σ is contained in $\mathcal{T}(B)$. That is, we constrain the values of the variables in C to be terms over $L(B)$. If our reduction ordering is oriented for (A, B), then $C \mid L(B)$ can be expressed as $C \mid a > U \wedge a > V \cdots$ where a is the least ground term not in $L(B)$, and $U, V, \ldots$ are the variables occurring in C. Thus, $C \mid L(B)$ is well-constrained. Finally, if S is a clause set, then let $S \mid L(B)$ stand for the set of $C \mid L(B)$ for C in S.

Lemma 2. *Let A and B be clause sets. If there is an interpolant for (A, B) in $\mathcal{L}_\forall$, then A and $B|L(B)$ are inconsistent.*

We are now ready to prove the key lemma that will allow us to build an interpolating prover. We show that on interpolation problems, superposition with procrastination makes only local deductions:

Lemma 3. *Let A and B be clause sets, and $\succ$ be a reduction order oriented for (A, B). Then system $\mathcal{I}_\succ$ with procrastination, applied to A and $B \mid L(B)$ generates only inferences local for (A, B).*

The above lemma holds only for provers that rigorously propagate the ordering constraints from one inference to the next. However, in practice this is not necessary to obtain a local proof. If we test the ordering constraints for satisfiability

but do not propagate them, the worst outcome is that unnecessary deductions will be made. We can simply throw away the resulting non-local deductions, since we know by the above lemma that they are not required for completeness.

Since saturation with procrastination is complete and generates local proofs for interpolation problems, we can use it to build an interpolation algorithm:

Algorithm 1

Input: A pair of equality clause sets (A, B) *having an interpolant in* $\mathcal{L}_\forall$.
Output: An interpolant for (A, B)
1) Choose a reduction order $\succ$ *oriented for* (A, B).
2) Apply system $\mathcal{I}_\succ$ *with procrastination to* $A, B \mid L(B)$.
3) If the prover generates a refutation P *local for* (A, B), *then*
4) Derive an interpolant for (A, B) *from* P *and output the result,*
5) Else (if the prover saturates) abort.

Theorem 2. *Algorithm 1 is correct and terminating.*

To allow us to speak of interpolants of program unwindings, we generalize the notion of interpolant from pairs to finite sequences of formulas. That is, an interpolant for a sequence of formulas $A_1, \ldots, A_n$ is a sequence $\hat{A}_1, \ldots \hat{A}_{n-1}$ such that:

- A_1 implies $\hat{A}_1$
- for all $1 \leq i < n$, $\hat{A}_i \wedge A_i$ implies $\hat{A}_{i+1}$
- $A_n \wedge \hat{A}_{n-1}$ implies false.
- for all $1 \leq i < n$, $\hat{A}_i \in (\mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n))$.

We can think of the interpolant for a sequence of formulas as being structured refutation of that sequence.

Though we do not prove it here, we can generalize Algorithm 1 to generate interpolants for sequences, replacing (A, B) with the sequence $A_1 \ldots A_n$. We say that a proof is local for $A_1 \ldots A_n$ when every inference is local to some A_i, and a reduction order $\succ$ is oriented for $A_1 \ldots A_n$ when it is oriented for all the pairs $(\{A_1 \ldots A_i\}, \{A_{i+1} \ldots A_n\})$. Finally, instead of $A, B \mid L(B)$, we refute $A_1, A_2 \mid L(A_2 \ldots A_n), \ldots, A_n \mid L(A_n)$. The result is a local refutation for $A_1 \ldots A_n$, from which we can derive an interpolant sequence in linear time in the proof size and n.

4 Invariant Generation

Now we come to the question of generating invariants with interpolants. The intuition behind this approach is the following. Suppose we wish to prove the correctness of a single-loop while program. For example, we might want to prove:

$$\{i = 0\} \ \textbf{while}\ i < N\ \textbf{do}\ a[i]{:=}0;\ i{+}{+}\ \textbf{od}\ \ \{\forall(0 \leq j < N)a[j] = 0\}$$

where $i{+}{+}$ is a shorthand for $i{:=}i + 1$. To do this, we might try unwinding the loop n times and proving the resulting in-line program. If we are lucky, the

resulting Floyd/Hoare proof will contain an inductive invariant for the loop. For example, for $n = 2$, we might have:

$$\begin{array}{lllll} \{i = 0\} & [i < N];\ a[i]{:=}0;\ i{+}{+} & \{\forall(0 \leq j < i)a[j] = 0\} \\ & [i < N];\ a[i]{:=}0;\ i{+}{+} & \{\forall(0 \leq j < N)a[j] = 0\} \end{array}$$

where $[\phi]$ denotes a guard. Note that the middle assertion of this proof is an inductive assertion for the loop, which we can verify with a suitable first-order prover. On the other hand, if we are unlucky, we might obtain:

$$\begin{array}{lllll} \{i = 0\} & [i < N];\ a[i]{:=}0;\ i{+}{+} & \{i = 1 \wedge a[0] = 0\} \\ & [i < N];\ a[i]{:=}0;\ i{+}{+} & \{\forall(0 \leq j < N)a[j] = 0\} \end{array}$$

This is also a valid proof, but the intermediate assertion is useless for generating an inductive invariant. If we unwind the loop further, we might obtain $i = 2 \wedge a[0] = 0 \wedge a[1] = 0$, and so on, producing a diverging series of non-inductive formulas.

As we will see, the Floyd/Hoare proofs for the unwindings can be produced by interpolation. The trick is to prevent the interpolant formulas from diverging as we unwind the loops further. We will show that by bounding the behavior of the prover appropriately, we can prevent divergence and guarantee to eventually produce an inductive invariant if one exists in $\mathcal{L}_\forall$.

Transition systems, unfoldings and interpolants. We will use first-order formulas to characterize the transition behavior of a system, using the usual device of primed symbols to represent the next state of the system. That is, a set of uninterpreted function and constant symbols S represents the system state. A state of the system is an interpretation of S. For every symbol $s \in S$, we let the symbol s' represent the value of s one time unit in the future. Moreover, we think of s with n primes added as representing the value of s at n time units in the future. For any formula or term ϕ, we will use the notation ϕ' to represent the result of adding one prime to all the occurrence of state symbols in ϕ (meaning ϕ at the next time), and $\phi^{\langle n \rangle}$ to denote the addition of n primes to all occurrence of state symbols in ϕ (meaning ϕ at n time units in the future).

A *state formula* is a formula in $\mathcal{L}(S)$ (which may also include various interpreted symbols, such as $\simeq$ and $+$). A *transition formula* is a formula in $\mathcal{L}(S \cup S')$. A *safety transition system* M is a triple (I, T, P), where state formula I represents the initial states, transition formula T represents the set of transitions, and and state formula P represents the set of safe states. A *safety invariant* for M is a state formula ϕ such that $I \models \phi$ and $\phi, T \models \phi'$ and $\phi \models P$. That is, a safety invariant is an inductive invariant of the system that proves that all reachable states satisfy P.

We will say that an *invariant generator* $\mathcal{G}$ is a procedure that takes a safety transition system M as input and outputs a sequence of formulas. For a given language $L \subseteq \mathcal{L}(S)$, we say that $\mathcal{G}$ is *complete for invariant generation* in L when, for every M that has a safety invariant in L, $\mathcal{G}$ eventually outputs a

safety invariant for M. If we have a complete invariant generation procedure, then we have an complete procedure to verify safety transition systems that have safety invariants in $\mathcal{L}_\forall$: we use a complete first-order prover to attempt to prove correctness of each invariant candidate in the sequence, in an interleaved manner.

Of course, there is a trivial complete invariant generator that simply outputs all of the formulas in L in order of their Gödel numbers. Our purpose here is to construct a *practical* invariant generator that uses proofs about finite behaviors to focus the invariant candidates on relevant facts, and thus in a heuristic sense tends to produce valid invariants quickly. In particular, we will be concerned with the language $\mathcal{L}_\forall(S)$ of universally quantified state formulas. We will describe a simple safety invariant generator based on our interpolation algorithm that is complete for invariant generation in $\mathcal{L}_\forall(S)$. It prevents divergence by bounding the language of the prover.

The algorithm is based on *unfolding* the transition system in the style of Bounded Model Checking [1]. For $k \geq 0$, the k-step unfolding of M (denoted $\mathcal{U}_k(M)$) is the following sequence of formulas:

$$\mathcal{U}_k(M) = I, T, T^{\langle 1 \rangle}, \ldots, T^{\langle k-1 \rangle}, \neg P^{\langle k \rangle}$$

This formulas characterizes the set of runs of the transition system of exactly k steps that end in an unsafe state. The system M is safe when $\mathcal{U}_k(M)$ is unsatisfiable for all $k \geq 0$. For simplicity, we will assume that $\neg P \wedge T \rightarrow \neg P'$. That is, once the safety condition is false, it remains false. This can easily be arranged by, for example, adding one state bit that remembers when the property has been false in the past.

To generate invariant candidates, we will make use of a *bounded* saturation prover to refute unfoldings. Given a language L, a saturation prover bounded by L simply throws away all derived clauses not in L (after attempting reductions). For example, the SPASS prover [14] implements bounding by W_k, the set of clauses with k symbols or fewer (*i.e.,* clauses of "weight" up to k). In the sequel, we will assume that $L_1, L_2, \ldots$ is a sequence of finite languages such that $L_1 \subset L_2 \subset \cdots$ and $\bigcup_i L_i$ is the set of all clauses. For example, the sequence $W_1, W_2, \ldots$ meets these criterion. Note that for any finite L, a saturation prover bounded by L must terminate on any input, since the number of clauses it can generate is bounded.

Now let $\succ$ be RPOS for some precedence order oriented for $S, S', S'', \ldots$ and let $\dot{\succ}$ be a reduction order such that the set of terms less than any given term t over a finite vocabulary is finite. For example, we could say that $t \dot{\succ} s$ when the weight of s is less than the weight of t or the weights are equal and $t \succ s$. Let SPB(L) stand for a saturation prover using the union of systems $\mathcal{I}_\succ$ and $\mathcal{I}_{\dot{\succ}}$ with procrastination, restricted to local deductions and bounded by L. For any system M with a universally quantified safety invariant, a fixed language L_m suffices to refute unfoldings of any length using this prover:

Lemma 4. *Let M be a safety transition system with a safety invariant in $\mathcal{L}_\forall$. There exists an integer m, such that for every $k \geq 0$, $SPB(L_m)$ refutes $\mathcal{U}_k(M)$.*

Our invariant generation algorithm is as follows (where $\text{unp}(\phi)$ is ϕ with primes removed):

Algorithm 2

Input: A system $M = (I, T, P)$ having a safety invariant in $\mathcal{L}_\forall$.
Output: A sequence of formulas containing a safety invariant for M.
1) Let $i = 1$ and $k = 1$
2) Repeat:
3) Apply Algorithm 1 using prover $SPB(L_i)$ to $\mathcal{U}_k(M)$.
4) If the algorithm returns an interpolant $\hat{A}$, then
5) For $j = 1$ to $k + 1$, output $\bigvee_{l=1\ldots j} unp(\hat{A}_l)$
6) Increase k.
7) Else (if Algorithm 1 aborts) increase i

Theorem 3. *Algorithm 2 eventually outputs a safety invariant for M.*

It is worth noting that this algorithm achieves completeness despite the fact that the prover is not "complete for consequence generation" as is required in [5]. The generated invariant candidates can be checked for inductiveness using any complete first-order prover. Since this prover may not terminate in the negative case, we must interleave these checks, rather than executing them sequentially. This is a fairly naïve approach to generating invariants from interpolants. We could also use, for example, the method of [7] or the lazy abstraction method [9]. Both of these methods would require a decision oracle for first-order logic. However, in practice we could use saturation of the (unbounded) prover as an indication of unsatisfiability and accept a possibility of non-termination.

5 Implementation and Experiments

The interpolation algorithm has been implemented by modifying the SPASS prover. This is an efficient saturation prover that implements superposition with a variety of reduction rules. SPASS was modified in several ways:

1. The procrastination rules were added.
2. The input formulas are numbered in sequence, and a precedence order is constructed that is oriented for that sequence.
3. Non-local inferences (after replacement) are discarded.

Moreover, it is also allowed to define a background theory by specifying additional interpreted function and predicate symbols and providing axioms for the theory. The background axioms may contain only interpreted symbols. Thus, use of the axioms does not affect locality of the proofs. When computing interpolants from proofs, the axioms in the proof are replaced with true, since they are tautologies of the theory.

The bound mechanism of SPASS was also modified to allow bounds other than weight (number of symbols) and nesting depth. In particular, we implemented a bounding scheme in which L_i allows all clauses with at most nesting depth i and at most i variables. This is a finite set of clauses (modulo subsumption).

For the experiments, we axiomatized three theories: the natural numbers with zero, successor and $<$, the theory of arrays with select and store, and transitive closure of arrays, with a reachability predicate. These axioms are necessarily incomplete. However, we found them adequate to prove properties of some simple programs manipulating arrays and linked lists. For each example program an assertion was specified. The the loops were manually unwound n times, for increasing values of n, and translated into static single-assignment (SSA) form in the manner of the CBMC tool [2]. These unwindings were then verified using the modified prover, increasing the bound i until a refutation was found for violation of the assertion. Then the interpolants were tested to see if they contain inductive invariants for the loops that prove the assertions.

Table 5 shows the results obtained. For each example, the table gives a brief description of the program, the assertion, the number of loop unwindings, the bound language required, and the run time of the prover in seconds.

Table 1. Results of invariant generation experiments

name	description	assertion	unwindings	bound	time (s)
`array_set`	set all array elements to 0	all elements zero	3	L_1	0.01
`array_test`	set all array elements to 0 then test all elements	all tests OK	3	L_1	0.01
`ll_safe`	create a linked list then traverse it	memory safety	3	L_1	0.04
`ll_acyc`	create a linked list	list acyclic	3	L_1	0.02
`ll_delete`	delete an acyclic list	memory safety	2	L_1	0.01
`ll_delmid`	delete any element of acyclic list	result acyclic	2	L_1	0.02
`ll_rev`	reverse an acyclic list	result acyclic	3	L_1	0.02

As an example, here is the (somewhat simplified) inductive invariant generated for example `list_acyc`. This is a loop in which newly allocated elements are added to the beginning of a list by modifying their link field:

```
and( reachable(link,x,nil),
     forall([U], or(U = nil, not(reachable(link,x,U)), alloc(U))))
```

This says that x (the list head) can reach nil (the list terminator) via the link field, and every cell reachable from x via the link field is allocated. The former condition guarantees that the list is acyclic, while the latter implies that in the future a cell already in the list will not be appended to the head, creating a

cycle. This shows one advantage of using interpolants for invariant generation relative to parametric invariant generation techniques such as [13]. That is, the interpolator is able to synthesize Boolean combinations without requiring the user to provide a template. Moreover, it can handle theories other than arithmetic, such as reachability. Using the interpolating superposition prover and lazy abstraction, the IMPACT software model checker [9] can automatically verify all of the above examples.

The linked list examples in the table could be handled easily by canonical heap abstraction methods [12]. However, using interpolation, we are not required to provide the instrumentation predicates that define the abstraction. This may be a significant advantage in scaling to larger programs. In the quantifier-free case at least, the ability of the interpolating prover to focus invariant generation on relevant facts has proved to be a significant advantage [7,4,9].

While the example programs we used are very simple, experience shows that even very simple programs can produce divergence in infinite-state verification techniques such as predicate abstraction [5]. Our results give some reason to believe that the divergence problem can be controlled.

6 Conclusion and Future Work

We have shown that, by a small modification of a paramodulation-based saturation prover, we can obtain an interpolating prover that is complete for universally quantified interpolants. This was done by constraining the reduction order and adding a reduction rule in order to obtain local proofs. We also solved the problem of divergence in interpolant-based invariant generation by bounding the language of the prover and gradually relaxing the bound. Some experiments verifying simple programs show that, in fact, divergence can be avoided, and termination can be achieved with shallow unwindings.

The next obvious task is to study the scaling behavior of the approach using a program verification system such as IMPACT [9], to determine whether the prover is capable of focusing on just the facts relevant to proving shallow properties of large programs. In addition, the are a number of possible extensions. The SPASS prover has the ability to split cases on ground atoms and to backtrack. However, it may still be much less efficient than a modern DPLL satisfiability solver. It might be useful to integrate it with an efficient DPLL solver in the style of "SAT modulo theories" (SMT) for greater efficiency. Moreover, it would be useful to integrate it with some ground arithmetic procedure (though again, the divergence problem would have to be solved).

Finally, it would be possible to use an interpolant generator for universally quantified interpolants as a predicate refinement heuristic for indexed predicate abstraction [6] much in the same way that this is done for ordinary predicate abstraction in [4]. Having an effective refinement heuristic might make the indexed predicate abstraction technique more practical.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, Springer, Heidelberg (1999)
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
3. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic 22(3), 269–285 (1957)
4. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM, New York (2004)
5. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
6. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 267–281. Springer, Heidelberg (2004)
7. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
8. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
9. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
10. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, ch. 7, pp. 371–443. Elsevier Science, Amsterdam (2001)
11. Robinson, G.A., Wos, L.T.: Paramodulation and theorem proving in first order theories with equality. Machine Intelligence 4, 135–150 (1969)
12. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL, pp. 105–118. ACM, New York (1999)
13. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)
14. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System description: SPASS version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)