A Practical and Complete Approach to Predicate Refinement

Ranjit Jhala and K. L. McMillan

[1] University of California, San Diego
[2] Cadence Berkeley Labs

Abstract. Predicate abstraction is a method of synthesizing the strongest inductive invariant of a system expressible as a Boolean combination of a given set of atomic predicates. A predicate selection method can be said to be complete for a given theory if it is guaranteed to eventually find atomic predicates sufficient to prove a given property, when such exist. Current heuristics are incomplete, and often diverge on simple examples. We present a practical method of predicate selection that is complete in the above sense. The method is based on interpolation and uses a "split prover", somewhat in the style of structure-based provers used in artificial intelligence. We show that it allows the verification of a variety of simple programs that cannot be verified by existing software model checkers.

1 Introduction

Predicate abstraction [14] is a technique commonly used in software model checking in which an infinite-state system is represented abstractly by a finite-state system whose states are the truth valuations of a chosen set of atomic predicates. The reachable state set of the abstract system corresponds to the strongest inductive invariant of the infinite-state system expressible as a Boolean combination of the given predicates.

Given a decision procedure for the underlying theory, predicate abstraction can prove a property of a system exactly when the property is implied by a quantifier-free inductive invariant of that system. That is, suppose that a system has a quantifier-free inductive invariant ψ that implies some condition that we wish to prove invariant and suppose we can supply the atomic predicates occurring in ψ. Since predicate abstraction synthesizes the strongest inductive invariant expressible as a Boolean combination of these predicates, it is guaranteed to generate an invariant as strong as ψ, and hence to prove the property. There remains only the question of how to guess these atomic predicates. We will say that a predicate selection heuristic is *complete* if it is guaranteed eventually to choose enough predicates to prove any given property ϕ, as long as there is some quantifier-free inductive invariant of the system that implies ϕ. This definition of completeness is strictly stronger than that of [1], which restricts invariants to atomic predicates generated by the "pre" operator.

There is, of course, a trivial complete heuristic. Since the atomic predicates are countable, one has only to enumerate them in a complete way. Each time we generate a new predicate, we add it to our set, and try predicate abstraction. Eventually our set of predicates will contain all the atomic predicates in ψ, and we will prove the property.

Obviously, this approach is not practical. Since predicate abstraction is exponential (or worse!) in the number of predicates, a practical approach must generate a sufficient set of predicates that is as small as possible. A number of heuristic approaches based on computing weakest preconditions have been suggested [8, 2, 3]. The approaches of [15, 7] derive predicates from proofs. None of these is complete in the above sense (though [8] includes an acceleration heuristic that may prevent divergence).

As an example of divergence of a predicate heuristic, consider the following simple C program fragment:[3]

```
x = i; y = j;
while (x!=0) {x--; y--;}
if (i == j) assert (y == 0);
```

A typical predicate heuristic will examine counterexamples produced by model checking the abstraction. A counterexample is a program execution path that reaches an error state, and cannot be refuted using the available predicates (a notion we will formally define later). Suppose our first counterexample passes through the loop zero times (which means $i = 0$ initially). We might obtain new predicates by computing the weakest precondition of the assertion in a backward manner along the path. From this we obtain formulas containing the atomic predicates $i = j$, $x = 0$, $y = 0$, $i = 0$ and $j = 0$. Using these predicates, we obtain a counterexample in which the loop is executed once. Computing weakest preconditions, we obtain the additional predicates $x = 1$, $y = 1$, $i = 1$, $j = 1$, and so on. Thus, by analyzing counterexamples, we obtain a diverging sequence of predicates in which the integer constants tend to infinity. On the other hand, the predicates $i = j$, $x = 0$ and $x = y$ are sufficient to prove the assertion (at the level of basic blocks). The loop invariant is $i = j \Rightarrow x = y$. Thus, predicate heuristics based on weakest precondition over counterexamples are incomplete, essentially due to a failure to generalize.

Heuristics based on interpolation [7] are potentially more effective in focusing on relevant predicates, but suffer from the same problem of divergence. In this paper, we propose a method that is both heuristically useful and complete (in the above limited sense). Like the method of [7], it is based on the computation of interpolants from the refutation of counterexamples. However, in this case the use of a specialized "split" prover allows us to restrict the language of the interpolants in a way that prevents the atomic predicates from diverging as the counterexamples become longer.

In the next section of the paper, we discuss the method of deriving predicates from interpolants, which are in turn derived from the refutation of counterex-

[3] Thanks to Anubhav Gupta for this example.

amples. We then show that by restricting the interpolants to a finite language L, and gradually expanding this language, we can guarantee convergence of predicate abstraction (when the property is provable). In section 3 we introduce the notion of a *split prover*, and show that such a prover can be used to generate interpolants in a restricted language, and thus can be used as a complete predicate heuristic. In section 4, we describe an implementation of such a prover for a particular theory. In section 5, we show that this method is capable in practice of verifying programs that cannot be verified by existing heuristic methods because of predicate divergence.

2 Predicates from interpolants

Throughout this paper, we will use standard first-order logic (FOL) and we will use the notation $\mathcal{L}(\Sigma)$ to denote the set of well-formed formulas (*wff 's*) of FOL over a vocabulary Σ of non-logical symbols. For a given formula or set of formulas ϕ, we will use $\mathcal{L}(\phi)$ to denote the *wff 's* over the vocabulary of ϕ.

For every non-logical symbol s, we presume the existence of a unique symbol s' (that is, s with one prime added). 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 $\phi^{\langle n \rangle}$ to denote the addition of n primes to every symbol in ϕ (meaning ϕ at n time units in the future). For any set Σ of symbols, let Σ' denote $\{s' \mid s \in \Sigma\}$ and $\Sigma^{\langle n \rangle}$ denote $\{s^{\langle n \rangle} \mid s \in \Sigma\}$.

Modeling programs We will use first-order formulas to characterize programs. To this end, let S, the state vocabulary, be a set of individual variables and uninterpreted n-ary functional and propositional constants. A *state formula* is a formula in $\mathcal{L}(S)$ (which may also include various interpreted symbols, such as $=$ and $+$). A *transition formula* is a formula in $\mathcal{L}(S \cup S')$. We require a technical condition: a transition formula must contain an occurrence of every symbol in S and S'. This condition can easily be made to hold by adding tautologies, such as $a = a$.

A *program* will be represented (somewhat abstractly) by a pair $(\mathcal{T}, \Pi)$ where $\mathcal{T}$ is a set of *transition formulas* (representing program statements) and $\Pi \subset \mathcal{T}^*$ is a regular language representing the possible execution paths of the program.

For any sequence of transitions $\pi = T_1, \ldots, T_n$ in $\mathcal{T}^*$, we will say the *unfolding* $\mathcal{U}(\pi)$ is the sequence $T_1^{\langle 0 \rangle}, \ldots, T_n^{\langle n-1 \rangle}$. For example, the unfolding of the error path of our example program that executes the loop once is:

$$x^{\langle 1 \rangle} = i^{\langle 0 \rangle} \wedge y^{\langle 1 \rangle} = j^{\langle 0 \rangle} \wedge i^{\langle 1 \rangle} = i^{\langle 0 \rangle} \wedge i^{\langle 1 \rangle} = j^{\langle 0 \rangle},$$

$$x^{\langle 1 \rangle} \neq 0 \wedge x^{\langle 2 \rangle} = x^{\langle 1 \rangle} - 1 \wedge y^{\langle 2 \rangle} = y^{\langle 1 \rangle} - 1 \wedge i^{\langle 2 \rangle} = i^{\langle 1 \rangle} \wedge j^{\langle 2 \rangle} = j^{\langle 1 \rangle},$$

$$x^{\langle 2 \rangle} = 0 \wedge i^{\langle 2 \rangle} = j^{\langle 2 \rangle} \wedge y^{\langle 2 \rangle} \neq 0$$

We will say that π is *feasible* when $\bigwedge \mathcal{U}(\pi)$ is consistent. We can think of a model of $\bigwedge \mathcal{U}(\pi)$ as a concrete program execution, assigning a value to every program variable at every time $0 \ldots n$. A program $(\mathcal{T}, \Pi)$ is said to be infeasible when every path in Π is infeasible. The problem of safety verification can be reduced to infeasibility by intersecting Π with the language of paths leading to "error" states.

Predicate abstraction Given a set of predicates β, we will say that *strongest β-postcondition* of a state formula ϕ with respect to transition T, denoted $\mathrm{sp}_T^\beta(\phi)$, is the strongest Boolean combination ψ over β such that $\phi \wedge T$ implies $\psi^{\langle 1 \rangle}$. That is, $\mathrm{sp}_T^\beta(\phi)$ is the strongest Boolean formula expressible over β that must be true after executing T from a state satisfying ϕ. We define the notion of strongest β-postcondition over sequences of transitions by induction over the sequence:

$$\begin{aligned} \mathrm{sp}_\epsilon^\beta(\phi) &= \phi \\ \mathrm{sp}_{\pi \cdot t}^\beta(\phi) &= \mathrm{sp}_t^\beta(\mathrm{sp}_\pi^\beta(\phi)) \end{aligned}$$

A sequence π of transitions is *β-refutable* when $\mathrm{sp}_\pi^\beta(\textsc{True}) \equiv \textsc{False}$. Further, a program $(\mathcal{T}, \Pi)$ is β-refutable when every path in Π is β-refutable. This is exactly the condition tested by predicate abstraction. That is, predicate abstraction can verify a program to be infeasible using predicates β exactly when the program is β-refutable. We will say that a *verifier* is a procedure that takes a program as input and returns TRUE or FALSE, or diverges. It is *sound* if it returns TRUE only when the program is infeasible. Moreover:

Definition 1. *A verifier $\mathcal{V}$ is* complete for predicate abstraction *if, for every program $\mathcal{A} = (\mathcal{T}, \Pi)$ that is β-refutable for some set β of atomic predicates, $\mathcal{V}$ converges on $\mathcal{A}$ and returns* TRUE.

Interpolants from 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 [5] states that an interpolant always exists for inconsistent formulas in FOL. To allow us to speak of interpolants of program paths, we generalize this idea to sequences of formulas. That is, given a sequence of formulas $\Gamma = A_1, \ldots, A_n$, we say that $\hat{A}_0, \ldots \hat{A}_n$ is an *interpolant* for Γ when

- $\hat{A}_0 = \textsc{True}$ and $\hat{A}_n = \textsc{False}$ and,
- for all $1 \leq i \leq n$, $\hat{A}_{i-1} \wedge A_i$ implies $\hat{A}_i$ and
- for all $1 \leq i < n$, $\hat{A}_i \in \mathcal{L}(A_i) \cap \mathcal{L}(A_{i+1})$.

That is, the i-th element of the interpolant is a formula in the common language of A_i and A_{i+1}, and is provable from the first i elements of Γ.

If Γ is quantifier-free, we can derive a quantifier-free interpolant for Γ from a refutation of Γ, in certain interpreted theories [11]. This fact was exploited in [7] to derive predicates for predicate abstraction. This is based on the following result, where $AP(\phi)$ denotes the set of atomic predicates in ϕ:

Theorem 1. *Given a set of atomic predicates β, a program path $\pi = A_1, \ldots, A_n$ is β-refutable* iff *$\mathcal{U}(\pi)$ has a quantifier-free interpolant $\hat{A}_0, \ldots, \hat{A}_n$ such that for all $1 \leq i < n$, $AP(\hat{A}_i) \subseteq \beta^{\langle i \rangle}$.*

Proof. For the *only if* direction, we observe that the sequence $\hat{P}_0, \ldots, \hat{P}_n$, where $\hat{P}_i = \mathrm{sp}^{\beta}_{A_1,\ldots,A_i}(\text{TRUE})^{\langle i \rangle}$, is a suitable interpolant (guaranteeing that $\hat{P}_i \in \mathcal{L}(A_{i-1}) \cap \mathcal{L}(A_i)$ requires our technical condition on transition formulas). For the *if* direction, we show by induction that P_i (as defined above) implies $\hat{A}_i$, hence $\mathrm{sp}^{\beta}_{\pi}(\text{TRUE}) \equiv \text{FALSE}$. □

If a counterexample path π is not β-refutable for the current set of predicates β we can compute an interpolant for π, and augment β by adding the atomic predicates occurring in the interpolant (dropping the primes). Thus, π is now β-refutable (for the new β). For example, a possible interpolant for the error path example above is

$$\text{TRUE}, (x^{\langle 1 \rangle} = i^{\langle 1 \rangle} \wedge y^{\langle 1 \rangle} = j^{\langle 1 \rangle}), (x^{\langle 2 \rangle} = i^{\langle 2 \rangle} - 1 \wedge y^{\langle 2 \rangle} = j^{\langle 2 \rangle} - 1), \text{FALSE}$$

From this we derive the predicates $x = i$,$y = j$,$x = i - 1$ and $y = j - 1$. This gives us a predicate heuristic (used in [7]) that is guaranteed to rule out any given counterexample path, but may produce predicates that diverge as the number of loop iterations increases. To prevent this divergence, we propose in this work to restrict the interpolants to some finite language L. For example, we could restrict L to contain numeric constants only in some fixed range, and thus prevent constants in the predicates from tending to infinity.

Definition 2. *Given a language L, an L-restricted interpolant for $\Gamma = A_1, \ldots, A_n$ is an interpolant $\hat{A}_0, \ldots \hat{A}_n$ for Γ, such that each $\hat{A}_i \in L$.*

By gradually enlarging the restriction language L, we obtain a complete procedure. That is, let us define a chain of finite, quantifier-free, propositionally closed languages $L_0 \subseteq L_1 \subseteq \cdots$ such that every atomic predicate is contained in some L_i. We can then use the following procedure for program verification:

procedure RELAX($\mathcal{A}$)
    let $k = 0$ and $\beta = \emptyset$
    repeat
        if $\mathcal{A}$ is β-refutable return TRUE
        else let π be a non-β-refutable path of $\mathcal{A}$ in
            if π has an L_k-restricted interpolant $\hat{A}_0, \ldots \hat{A}_n$
            then let $\beta = \beta \cup \{p \mid p^{\langle i \rangle} \in AP(\hat{A}_i),\ \text{for some } 1 \leq i < n\}$
            else let $k = k + 1$

That is, as long as the counterexamples produced by predicate abstraction are refutable using predicates in language L_k we continue using L_k-restricted interpolants to generate predicates. When we obtain a counterexample not refutable in L_k, we move on to L_{k+1}.

Theorem 2. *Procedure* RELAX *is complete for predicate abstraction.*

Proof. Theorem 1 tells us that each interpolant must contain some atomic predicate not in β. Thus in every iteration of the loop, either β increases, or k increases. Since $\beta \subseteq L_k$, it cannot increase unboundedly without increasing k. Thus either the procedure terminates, or k increases unboundedly. Now suppose program $\mathcal{A}$ is β-refutable. β must be contained in some L_m. We know k cannot increase beyond m, since by Theorem 1, every path of $\mathcal{A}$ has an L_m-restricted interpolant. Thus the procedure terminates (and returns TRUE). $\square$

Of course, the choice of restriction languages L_k is a heuristic one, and we would like to make that choice in a way that will lead to rapid convergence. One observation we can make in this area is that invariants of loops rarely contain large numeric constants. Thus, we might define L_k so as to contain numeric constants no larger in absolute value than k. This heuristic is effective for our example program. Note that L_0 does not contain the interpolant we obtained above for our example path (since it contains the constant 1). Thus, it forces us to choose an interpolant like this: TRUE,$(x^{\langle 1\rangle} = i^{\langle 1\rangle} \wedge y^{\langle 1\rangle} = j^{\langle 1\rangle})$, $(i^{\langle 2\rangle} = j^{\langle 2\rangle} \Rightarrow x^{\langle 2\rangle} = y^{\langle 2\rangle})$, FALSE. This yields predicates $x = i$,$y = i$,$x = j$ and $x = y$, which are adequate to prove the program. One way to view this is that the inability to use the specific constant 1 forces us to generalize. Thus, the verification terminates at L_0.

3 The split prover

The problem of predicate selection has now been reduced to finding an L-restricted interpolant for a given sequence of formulas $A_1, \ldots, A_n$. As in [11] we derive interpolants from proofs. However, we restrict the interpolants to formulas in L by placing a restriction on allowable proofs. We define a notion of *split proof* in which all reasoning is local. That is, each deduction step is labeled by some A_i, such that both its antecedents and consequent are contained in $\mathcal{L}(A_i)$, and the deduction depends only on A_i. This is as if we have n communicating provers, each of which knows one A_i, and can see the results of other provers only if they are over the vocabulary of A_i. To be more precise:

Definition 3. *A split proof over a set of hypotheses Γ is a triple (V, E, P), such that V is a set of formulas, (V, E) is a directed acyclic graph, and P is a labeling function $V \to \Gamma$, and*

- *for all edges $(g, f) \in E$, we have $g, f \in \mathcal{L}(P(f))$, and*
- *$preds(f), P(f) \models f$.*

where $preds(f) = \{g \mid (g, f) \in E\}$. A split refutation of Γ is a split proof over Γ whose unique leaf is the formula FALSE.

In order to restrict the interpolants to a given language L, we have only to restrict the set of formulas that can be communicated between provers:

Definition 4. *An L-restricted split proof over a set of hypotheses Γ is a split proof (V, E, P) over Γ, such that, for all edges $(f, g) \in E$, if $P(f) \neq P(g)$, then $f \in L$.*

We will say that a sequence of hypotheses $\Gamma = \{A_1, \ldots, A_n\}$ is *strict* if the vocabularies of A_i and A_j only intersect when $i - 1 \leq j \leq i + 1$ (*i.e.*, if only nearest neighbors share non-logical symbols). This condition is satisfied by program path unfoldings. We can now show the following:

Theorem 3. *Given a strict sequence of hypotheses $\Gamma = \{A_1, \ldots, A_n\}$, and a propositionally closed language L, Γ has an L-restricted interpolant if and only if it has an L-restricted split refutation.*

Proof. The *only if* is straightforward, since the interpolant itself acts as the refutation. That is, each formula in the interpolant proves the next, given the next hypothesis, and each is over the common language of neighboring hypotheses. For the *if* direction, we construct an interpolant from the proof as follows. First, we rewrite the proof so that every edge is between vertices with distinct labels. If an edge (f, g) is such that $P(f) = P(g)$, we eliminate it by adding preds(f) to preds(g). Now we transform each vertex f into the formula $f' = \bigwedge \text{preds}(f) \Rightarrow f$. We then create new strict hypotheses $\Gamma' = \{A'_1, \ldots, A'_n\}$ where $A'_i = \{f' \mid P(f) = A_i\}$. Note that A_i implies A'_i and $A'_i \in \mathcal{L}(A_i)$, by Definition 3, and $A'_i \in L$, by Definition 4. This set of formulas is propositionally unsatisfiable (*i.e.*, no truth assignment to the atoms makes it true). Therefore, we can construct a propositional interpolant for it, without introducing new atoms, by the method of [10] (this step requires strictness). Since each A_i implies A'_i, it follows that this is also an interpolant for $A_1, \ldots, A_n$, and moreover it is L-restricted. □

The proof of this theorem also gives us a procedure for constructing an L-restricted interpolant from an L-restricted split proof. We transform the proof into a sequence of formulas, refute this sequence propositionally, and then derive the interpolant from the propositional refutation. This is actually a polynomial-time operation, since the refutation can be done by unit resolution (*i.e.*, BCP).

The key question is how to find a suitable split proof. Proofs generated by an arbitrary prover will not in general fit our restrictions. Interestingly, this question has been studied in the artificial intelligence community, for the purpose not of generating interpolants, but of creating more efficient provers by localizing the proof effort. The method is based on the notion of *consequence finding*.

A consequence finder is a function that takes a set of hypotheses Γ in its input language and generates a set of consequences of Γ. For a given language L, we will say that a consequence finder $\mathcal{R}$ is *complete for L-generation* when every consequence of Γ in L is implied by $\mathcal{R}(\Gamma)$. That is, to be complete $\mathcal{R}$ need not generate *every* consequence of Γ in L, but it must preserve all consequences of Γ expressible in L. To be more formal:

Definition 5. *A* consequence finder $\mathcal{R}$*, with* input language $\mathcal{L}(\mathcal{R})$ *is a function* $\mathcal{P}(\mathcal{L}(\mathcal{R})) \to \mathcal{P}(\mathit{wff})$ *that is monotone,* $\cup$*-continuous, and such that, for every* $\phi \in \mathcal{R}(\Gamma)$, $\Gamma \models \phi$.

Definition 6. *Given a language* L*, a* consequence finder $\mathcal{R}$ *is* complete for L-generation iff*, for every formula* $f \in L$*, if* $\Gamma \models f$*, then* $\mathcal{R}(\Gamma) \cap L \models f$.

We now use this notion of consequence finding to build a prover that constructs split proofs. For each partition A_i, we construct a consequence finder $\mathcal{R}_i$ that is complete for $\mathcal{L}(A_{i-1})$-generation and for $\mathcal{L}(A_{i+1})$-generation. In other words, each prover can generate all consequences in the languages of its neighbors. The initial input of $\mathcal{R}_i$ is just A_i. Each time a consequence ϕ is generated by some $\mathcal{R}_i$, it is added to the input of every $\mathcal{R}_j$ such that $\phi \in \mathcal{L}(A_j)$. We can formalize this notion of a combination of local consequence finders as follows:

Definition 7. *Let* $\boldsymbol{\mathcal{R}}$ *be an indexed set of consequence finders* $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$*. The* composition *of* $\boldsymbol{\mathcal{R}}$*, denoted* $\otimes\boldsymbol{\mathcal{R}}$*, is a function that takes an indexed set of hypotheses* $\Gamma = \{A_1, \ldots, A_n\}$*, such that* $\mathcal{L}(A_i) \subseteq \mathcal{L}(\mathcal{R}_i)$*, and returns the least fixed point of function* $\mathcal{F}$*, where*

$$\mathcal{F}(Q) = \bigcup_i \mathcal{R}_i(A_i \cup (Q \cap \mathcal{L}(\mathcal{R}_i)))$$

Note that the monotonicity of consequence finders guarantees the existence of the least fixed point of $\mathcal{F}$ in the above definition. In [9] it is shown (in a somewhat more general setting) that such a split prover is complete for refutation in FOL:

Theorem 4 ([9]). *Let* $\Gamma = \{A_1, \ldots, A_n\}$ *be a strict sequence of hypotheses in FOL, and let* $\boldsymbol{\mathcal{R}} = \mathcal{R}_1, \ldots, \mathcal{R}_n$ *be an indexed set of consequence finders, such that* $\mathcal{L}(\mathcal{R}_i) = \mathcal{L}(A_i)$*. If for every* $1 \leq i < n$*,* $\mathcal{R}_i$ *is complete for* $\mathcal{L}(A_{i+1})$*-generation, then* $\textsc{False} \in (\otimes\boldsymbol{\mathcal{R}})(\Gamma)$ iff Γ *is inconsistent.*

This is a simple consequence of Craig's interpolation lemma. That is, assuming $\mathcal{R}_i$ receives enough facts to prove component $\hat{A}_i$ of the interpolant, it produces enough facts to imply $\hat{A}_{i+1}$. Thus we must derive $\hat{A}_n$, which is false.

In [9], various resolution strategies are discussed which are complete for $\mathcal{L}(\Sigma)$-generation in FOL, where Σ is an arbitrary vocabulary of non-logical symbols. This makes it possible to construct a complete split prover for FOL. Our concern, however, is not to implement a complete prover, but rather a prover that is "complete" for generation of L-restricted split proofs, for a particular language L. Our approach to this is to restrict communication between consequence finders to just sentences in the restriction language L:

Definition 8. *Let* $\boldsymbol{\mathcal{R}}$ *be an indexed set of consequence finders* $\mathcal{R}_1, \ldots, \mathcal{R}_n$*, and let* L *be a language. The* L-restricted composition *of* $\boldsymbol{\mathcal{R}}$*, denoted* $\otimes^L\boldsymbol{\mathcal{R}}$*, is a function that takes an indexed set of hypotheses* $\Gamma = \{A_1, \ldots, A_n\}$*, such that* $\mathcal{L}(A_i) \subseteq \mathcal{L}(\mathcal{R}_i)$*, and returns the least fixed point of function* $\mathcal{F}$*, where*

$$\mathcal{F}(Q) = \bigcup_i \mathcal{R}_i(A_i \cup (Q \cap \mathcal{L}(\mathcal{R}_i) \cap L))$$

We can show that the L-restricted split prover defined above is complete for generation of L-restricted split refutations, provided the consequence finders are complete for L-generation:

Theorem 5. *Let $\Gamma = \{A_1, \ldots, A_n\}$ be a strict sequence of hypotheses in FOL, and let $\boldsymbol{\mathcal{R}} = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ be an indexed set of consequence finders, such that $\mathcal{L}(\mathcal{R}_i) = \mathcal{L}(A_i)$, and let L be an arbitrary language. If for each $1 \leq i < n$, $\mathcal{R}_i$ is complete for $(\mathcal{L}(A_{i+1}) \cap L)$-generation, then* FALSE $\in (\otimes^L \boldsymbol{\mathcal{R}})(\Gamma)$ iff *Γ has an L-restricted split refutation.*

Proof. For the *if* direction, if Γ has an L-restricted split refutation, it has an L-restricted interpolant $\hat{A}_0, \ldots, \hat{A}_n$ (by Theorem 3). By induction on i, each $\mathcal{R}_i$ must generate facts implying $\hat{A}_i$, since it is complete for $(\mathcal{L}(A_{i+1}) \cap L)$-generation. Thus $\mathcal{R}_n$ generates FALSE. For the *only if* direction, we show by induction that each fact $f \in (\otimes \boldsymbol{\mathcal{R}})(\Gamma)$ has an L-restricted split proof. The function $\mathcal{F}$ is a finite union of $\cup$-continuous functions and so is $\cup$-continuous. Therefore, by the Tarski-Knaster theorem f must occur in some fixed point iteration $\mathcal{F}^j(\emptyset)$. Thus f is a consequence of some facts $\Theta \subseteq \mathcal{F}^{j-1}(\emptyset)$, generated by some $\mathcal{R}_i$. By inductive hypothesis, these facts have L-restricted split proofs. Thus f has an L-restricted split proof. □

An immediate consequence is, of course, that the L-restricted split prover generates a proof exactly when Γ has an L-restricted interpolant, and moreover this proof can be translated directly into an interpolant (Theorem 3).

Note that for completeness the fixed point of Definition 8 need not converge finitely – the prover may generate consequences infinitely if no L-restricted refutation exists. One could still obtain a complete verifier by trying all the restriction languages L_k in parallel. In practice, of course, we want to obtain a negative result quickly so we can advance to the next L_k.

4 Implementing a split prover

In this section we describe an attempt to implement an efficient split prover for a limited theory. Our *wff's* are limited to quantifier-free first-order sentences, with equality, separation predicates (difference bounds), and restricted use of the array operators "select" and "store". The only arithmetic predicates allowed in the theory are of the form $x - y \leq c$, $x \leq c$ and $c \leq x$, where c is an integer constant. This simple theory appears to be sufficient to handle many properties of programs that manipulate arrays. The prover is complete for split proof generation for rational models, but not (yet) for integer models. The prover generates interpolants in a restriction language L. This language is defined by a finite set C_D of constants that may occur in difference bounds of the form $x - y \leq c$, and a finite set C_B of constants that may occur in absolute bounds of the form $x \leq c$ or $x \geq c$. To make L finite, we also require a bound b_f on the nesting depth of uninterpreted function symbols.

There is not space here to discuss all of the issues involved in constructing an efficient prover. Rather, we will give an informal overview of the main features of the prover, with emphasis on the issues that differentiate a split prover from a non-split prover. For efficiency, we separate the propositional reasoning from the theory reasoning. Propositional reasoning is handled by an efficient Boolean satisfiability (SAT) solver, similar to Chaff [12]. We construct complete consequence finders for the several theories, and combine them using the Nelson Oppen approach [13]. As in that method, we rely on convexity of the theories to avoid generating disjunctions, and we split cases when necessary to eliminate non-convexities.

Coupling of propositional and theory reasoning is done in the "lazy" manner, as in [6]. The SAT solver tests whether the entire set of hypotheses $\Gamma = \{A_1, \ldots, A_n\}$ is propositionally consistent. If so, it produces a propositional satisfying assignment as a set of literals $\mathcal{W}$. This set is partitioned into subsets $\{W_1, \ldots, W_n\}$, such that each atom of W_i occurs in A_i. This set of hypotheses is then passed to the split theory prover for refutation. The hypotheses used in the generated refutation are collected, and their dual is passed to the SAT solver as a "blocking clause" – a tautology that rules out the given satisfying assignment. The process continues until either the system becomes propositionally unsatisfiable (and thus Γ is refuted) or until some propositional satisfying assignment cannot be refuted. The propositional decision procedure need not be "split". Rather the propositional proof and the split proofs of the blocking clauses can be combined as in [11]. The interpolant derived from this combined proof is still guaranteed to be L-restricted, since the propositional interpolation rules do not generate new atoms.

By separating propositional and theory reasoning in this way, we limit the hypotheses of the split prover to just sets of literals. This greatly simplifies consequence generation. In particular, it allows us to take advantage of the *convexity* of a given theory, as in [13]. We will say that a complete consequence finder is convex if it generates only Horn clauses (clauses with at most one positive literal). If our consequence finders are convex, then we can further restrict the language L by which the provers communicate to contain only positive literals. This is because unit resolution is complete for Horn clause refutation (and is exploited in the Nelson Oppen method).

Difference bounds The problem is thus reduced (in the convex case) to one of "unit" consequence finding in the given theory, in the given language L. We will begin with the theory of difference bounds (considering rational models first). For this theory, we use the linear combination rule to generate consequences. This derivation rule takes as antecedents two inequalities $0 \leq x$ and $0 \leq y$ (where x and y are arbitrary terms) and derives an inequality $0 \leq c_1x + c_2y$, where c_1 and c_2 are positive constants. We restrict use of the rule to just cases where the consequent is a difference bound. For example, from $x \leq y + 2$ and $y \leq z + 3$, we can derive $x \leq z + 5$. We can discard consequences that are subsumed by previously generated consequences, and we terminate if a contradiction (say, $0 \leq -1$) is derived. This rule is complete for consequence generation over a given vocabulary (applying it exhaustively amounts to an all-pairs shortest path computation on a graph whose vertices are terms and whose edges are labeled with difference bounds).

The rule is not complete, however, for consequence generation in our restriction language L. Consider the case, for example, where we can derive $x \leq y - 1$, but the set C_D of allowed difference constants is just $\{0\}$. Thus $x \leq y - 1$ is not in L, but its consequence $x \leq y$ *is* in L. Unfortunately, $x \leq y$ cannot be derived by the linear combination rule. To remedy this, we add a weakening rule, that derives from an inequality $0 \leq x$ a weaker inequality $0 \leq x + c$, where c is a positive constant. This rule is used to derive the strongest consequence of any inequality that is contained in L. With this rule, and similar rules for strict inequalities, our system is complete for L-generation over the rationals.

Equality and uninterpreted functions Next, we consider equality and uninterpreted function symbols. For this theory, we use the usual derivation rules for equality: symmetry, reflexivity, transitivity and congruence, along with the contradiction rule (any literal and its negation imply FALSE). These rules are complete for unit consequence finding. They are not, however, finitely terminating, because of the congruence rule. For example, given $a = b$, we will derive $f(a) = f(b)$, $f(f(a)) = f(f(b))$, and so on. Though termination is not necessary for completeness, it is, of course, desirable in practice. For purposes of refutation, we can force termination by restricting the congruence rule to generate only terms that occur in the hypotheses. This is done in the usual congruence closure approach (completeness of this approach is another consequence of Craig's interpolation lemma). However, this method is not complete for consequence finding. Suppose, for example, we have the hypotheses $a = b$ and $f(a) = c$, and $L = \mathcal{L}(\{b, c, f\})$. There are no consequences in this language over just the terms $a, b, c, f(a)$. However, $f(b) = c$ is derivable. We can remedy this deficit by allowing the congruence rule to derive equalities over any terms occurring in the hypotheses *or* having function nesting depth within our bound b_f on function nesting. Since this is a finite set of terms, our rules are now terminating, and complete for L-consequence generation.

We combine difference bound and equality reasoning in the manner of Nelson and Oppen [13]. That is, we compose two consequence finders, one for difference bounds and one for equality. For difference bounds, complete consequence generation in the language of equality is achieved by a rule that derives $a = b$ from $a \leq b$ and $b \leq a$. For equality, complete consequence generation in the language of difference bounds is obtained by a rule that derives $a \leq b$ and $b \leq a$ from $a = b$.

The theory of arrays The first-order theory of arrays provides two interpreted functions select and store. The term $\text{select}(a, n)$ represents the n-th element of array a, while $\text{store}(a, n, b)$ is the array resulting from setting the n-th element of array a to value b. These functions obey the following axioms:

$$\text{select}(\text{store}(a, n, b), n) = b$$

$$n \neq n' \Rightarrow \text{select}(\text{store}(a,n,b),n') = \text{select}(a,n')$$

The second axiom is problematic on two counts. First, it generates an infinite set of quantifier-free consequences. For example, if we have the hypothesis $a' = \text{store}(a,n,b)$, then we can derive $\text{select}(a',n+1) = \text{select}(a,n+1), \text{select}(a',n+2) = \text{select}(a,n+2), \ldots$. There is one such consequence for every term provably not equal to n. Although there is a finite number of such terms occurring in L, enumerating them all would still be extremely inefficient. However, we can avoid this difficulty by restricting the use of arrays. That is, we allow array-valued terms to occur in A_i only as the first argument of select and in expressions of the following form:

$$a' = \text{store}(\text{store}(\ldots \text{store}(a,n_1,b_1),n_2,b_2)\ldots,n_k,b_k)$$

where a' does not occur in A_{i-1} and a does not occur in A_{i+1}. This corresponds to the way in which arrays are used in imperative programs (that is, once the array is modified, the old value of the array is no longer accessible). In this case, it suffices to instantiate the second array axiom only for terms n' that occur as array indices in some select or store term (as no consequences are possible for other array indices in $\mathcal{L}(A_{i-1})$ or $\mathcal{L}(A_{i+1})$).

The second problem is the non-convexity of the array theory. That is, the second axiom is in effect a disjunction of positive literals. In the case where we cannot infer the truth value of either literal in the disjunction, and we cannot otherwise obtain a refutation, we simply abandon the proof and introduce the clause $(n = n' \vee n \neq n')$ into the SAT solver, causing it to decide the value $n = n'$, and thus eliminate the non-convexity. This tactic is also used in various "lazy" decision procedures.

Integer models For program verification, we need to interpret formulas over integer models. This is problematic, since integer difference-bound arithmetic is non-convex when L includes equality formulas. In fact, deciding consistency of a set of literals in this theory is already NP-complete. Moreover, our additional restrictions on L introduce additional non-convexity. For example, suppose that $C_D = \{0\}$ and we have $x \leq j$ and $y \leq x + 1$. The disjunction $x \geq y \vee y \leq j$ is a consequence, but is not implied by any unit consequence in L. At this point we have not attempted to tackle the problem of a complete and heuristically efficient split prover for integers. Rather, we have added two simple rules that seem to be adequate in most practical cases for programs manipulating arrays. The first derives $a \geq b + 1$ from $\neg(a \leq b)$ and the second derives $a \geq b + 1$ from $a \geq b$ and $a \neq b$.

Unfoldings in SSA form A common optimization used, *e.g.*, in [7] is to write the unfolding of a program path in the more compact "static single-assignment" form (SSA). This can also be done with the split prover, if we relax our requirement of strictness in the unfolding (*i.e.*, that each time frame shares symbols only with its nearest neighbors). This complicates the above theory, but does not

Outcome	SatAbs	Magic	Blast (old)	Blast (new)
Verified	0	0	8	12
Refinement failed	13	13	0	0
Did not finish	0	0	5	1

Table 1. Outcomes on test programs

present any difficulty in practice. Further, since in this scheme a fact may be deduced by many consequence finders, we adopt an approach in which such a fact is deduced only once, and its derivation labeled with the range of time frames in which it is deduceable. Thus, we can more efficiently handle long unfoldings.

5 Experiments

To test the split prover as a predicate heuristic, we wrote a collection of small C programs containing loops and decorated with assertions.[4] The assertions are all provable by quantifier-free invariants, and thus by predicate abstraction. All require non-trivial invariants, in the sense that the loop index variable(s) must occur in the invariants. Most of the programs perform operations on arrays or zero-terminated C strings, such as filling, copying, concatenating and substring extraction. We tested four predicate abstraction tools on these programs: SatAbs [4], Magic [3], Blast [7] without the split prover (old), and Blast with the split prover (new). The outcomes are tabulated in Table 1. SatAbs and Magic, whose predicate heuristics are based on weakest preconditions, are unable to verify any of the 13 programs.[5] In all cases, the predicate refinement step fails to produce new predicates at some point (except for three cases in which Magic incorrectly produces counterexamples). Blast, whose predicate heuristic is based on interpolation, verifies 8 of the 13 examples without using the split prover. On the remaining 5, the constants in the predicates diverge to infinity (or toward some intractably large upper bound).

For the split prover version of Blast, we define the restriction language L_k by $C_D = \{-k, \ldots, k\}$ and $C_B = \{c + d \mid c \in C_P,\ d \in C_D\}$, where C_P is the set of numeric constants occurring in the program. That is, we allow difference bounds nearby to zero, and absolute bounds (such as $x \leq c$) nearby to some constant occurring in the program. The latter are useful for loops whose upper or lower bounds are fixed constants. As we increase k, we gradually expand the set of available constants until an inductive invariant can be expressed. For these programs, we do not require a limit b_f on function symbol nesting, since no functions are iterated (we might require a limit, for example, if the programs traversed linked lists). Using this heuristic, we find that 12 of the 13 programs can be verified. All successful runs complete in under 30 seconds. In one case,

[4] Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil`.

[5] For SatAbs, we used version 1.1 with default settings. For Magic, we used version 1.0 with `--optPred --predLoop 2`.

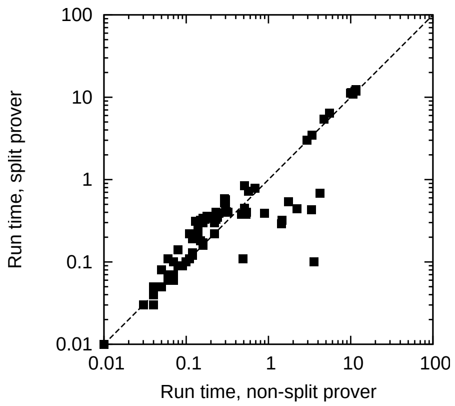

Fig. 1. Run time comparison of split and unsplit provers

we time out because a loop requires the invariant $i \leq j <= 200$, which does not occur until L_{200}. In this case, it appears that our notion of C_D requires some adjustment – perhaps allowing difference bounds nearby the large constants in the program.

To test the performance of the split prover, we compare it with the non-split interpolating prover of [11], which uses a conventional Nelson Oppen procedure for theory reasoning. Figure 1 plots run times in seconds for the set of unfoldings generated in verifying the two largest device driver examples from [7], with the split prover restricted to L_0. Two unfoldings that could not be refuted using L_0 were removed. Each point represents one unfolding. It can be seen that the split prover is only slightly less efficient than the unsplit prover.

6 Conclusion and future work

Existing predicate heuristics are incomplete, in that they may fail to find an adequate set of predicates when one exists. However, by restricting the predicates to a finite set, and progressively relaxing this restriction, we can obtain a complete method. In an interpolant-based approach, this can be done using a "split prover" that restricts the language of communication between time frames. We have shown that a practical split prover can be built, at least for difference bound arithmetic over the rationals. Moreover, a suitable choice of restriction language allows us to verify programs for which existing methods fail in practice. Thus, we have a predicate heuristic that is both theoretically complete and practically useful. For future work, it would be useful to expand the prover beyond difference bound arithmetic (though it is not clear what a suitable restriction language would be in this case) and to handle additional theories, such as the theory of bit vectors.

The main limitation of the method is a limitation of predicate abstraction itself, which cannot synthesize quantified invariants. For example, consider the following simple C program:

```
for(i = 0; i < n; i++) x[i] = 0;
for(i = 0; i < n; i++) assert(x[i] == 0);
```

An invariant for this program requires a quantifier. Though in principle predicate abstraction can use quantified predicates, they must be provided by the predicate heuristic – predicate abstraction cannot synthesize them from atomic formulas. The next step in this work is to produce quantified predicates. Some preliminary results have been obtained in this area. For example, by removing the restriction that interpolants be quantifier-free, we can obtain sufficient quantified predicates to verify the above program (including the invariant for the first loop $\forall j.\ (0 \leq j < i) \Rightarrow x[j] = 0$).[6] Ultimately the goal is to extend the range of predicate abstraction to a richer class of programs and properties.

Acknowledgment. The authors thank Tal Lev-Ami for pointing out related work in Artificial Intelligence.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, pages 158–172, 2002.
2. T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in c programs. Technical Report MSR-TR-2002-09, Microsoft, 2002.
3. S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME*, pages 19–34, 2003.
4. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.
5. W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic*, 22(3):269–285, 1957.
6. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE*, pages 438–455, 2002.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
8. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS*, pages 98–112, 2001.
9. S. McIlraith and E. Amir. Theorem proving in structured theories (full report). Technical Report KSL-01-04, Stanford, 2001.
10. K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.
11. K. L. McMillan. An interpolating theorem prover. In *TACAS*, pages 16–30, 2004.
12. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.
13. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Prog. Lang. and Sys.*, 1(2):245–257, 1979.
14. H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
15. R. Majumdar T. A. Henzinger, R. Jhala and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[6] Thanks to Daniel Kröning for integrating this in SatAbs.