Lazy Abstraction with Interpolants

Kenneth L. McMillan

Cadence Berkeley Labs

Abstract. We describe a model checker for infinite-state sequential programs, based on Craig interpolation and the lazy abstraction paradigm. On device driver benchmarks, we observe a speedup of up to two orders of magnitude relative to a similar tool using predicate abstraction.

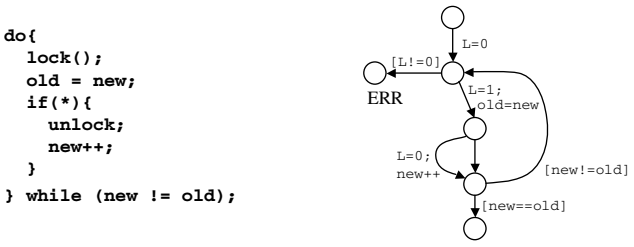
1 Introduction

Craig interpolants derived from proofs have been shown to provide an efficient method of image approximation in finite-state symbolic model checking [10]. In this paper, we extend the interpolation-based model checking approach from finite- to infinite-state systems, in particular to the verification of sequential programs. The approach applies an interpolating prover [11] in the lazy abstraction paradigm [7]. Instead of iteratively refining an abstraction, lazy abstraction refines the abstract model on demand, as it is constructed. Up to now, this refinement has been based on predicate abstraction [12]. Here, we refine the abstraction using interpolants derived from refuting program paths. This avoids the high cost of computing the predicate image (or abstract "post") operator, yielding a substantial performance improvement.

To illustrate the algorithm, we will use the simple C fragment of Figure 1 (borrowed from [7]). We model the functions `lock` and `unlock` by setting and resetting a variable `L` representing the state of the lock. We would like to prove that `L` is always zero on entry to `lock`. A control-flow graph for the function is shown in the figure. We have initialized `L` to zero and added a transition to an error state when `lock` is called and `L` is non-zero. Our algorithm unwinds the control-flow graph of the program into a tree. Each vertex in the tree corresponds to a program control location, and is labeled with a fact about the program variables that is true at that point in the execution of the program. Each vertex is initially labeled TRUE. When we reach a vertex corresponding to the error location, we strengthen the facts along the path to that vertex, so as to prove the error vertex unreachable.

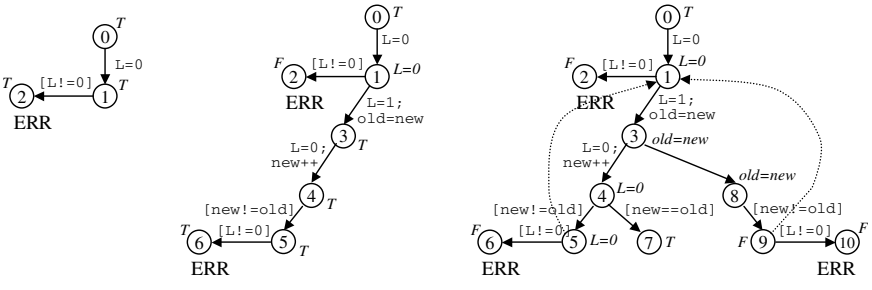
For example, suppose we first expand the path that branches to the error location on entering the loop (Figure 2a). We wish to label the error vertex FALSE, thus proving it unreachable. This is done by generating an *interpolant* for the path to the error state. An interpolant for a path is a sequence of formulas assigned to the vertices, such that each formula implies the next after executing the intervening program operation, and such that the initial vertex is labeled TRUE and the final vertex FALSE. Existence of an interpolant implies

T. Ball and R.B. Jones (Eds.): CAV 2006, LNCS 4144, pp. 123–136, 2006.



(a) program fragment (b) control-flow graph

Fig. 1. A simple example program



(a) first error (b) second error (c) termination

Fig. 2. Stages of the unwinding (vertex labels in italics)

that the final (error) vertex is unreachable. An interpolant can be derived from a refutation of the path generated by a theorem prover [11,6]. In Figure 2a, an interpolant would be: TRUE,$L = 0$,FALSE. In Figure 2b, we have strengthened the labeling on the error path with this interpolant (ruling out the error) and backtracked to explore the non-error branch. We pass through the loop, calling `lock` and `unlock`, then return to the top, taking the error branch again. In this case, our interpolant labels vertices 4 and 5 with $L = 0$ (again labeling the error vertex FALSE). Notice that vertices 5 and 1 correspond to the same location (the top of the loop) and that the label of vertex 5 implies the label of vertex 1. We say that vertex 1 *covers* vertex 5, and we cease expanding descendants of the covered vertex. However, if vertex 1 were to be strengthened in the future, it might cease to cover vertex 5, and we would have to continue expanding it.

Figure 2c shows the remainder of the unwinding, indicating coverings with dotted lines. We backtrack, expanding the path that falls out of the loop, and then the path that skips the call to `unlock`. In the latter case, we again reach an error state, strengthening the path. This labels vertex 9 with FALSE, thus it is also covered by vertex 1. At this point all unexpanded states are covered, so the procedure terminates. At termination, the disjunction of the labels for a given location is an invariant for that location. Notice also that the labels use the atomic predicates $L = 0$ and $old = new$, but are not the strongest facts expressible using those predicates (as we would obtain with predicate abstraction).

Rather, they are just strong enough to allow us to label the error vertices FALSE. Notice that we could also strengthen a path by computing strongest postconditions or weakest preconditions along the path (these are, in fact, the strongest and weakest interpolants respectively). However, by deriving interpolants from proofs, we exploit the prover's ability to focus on relevant facts, and thus avoid deducing irrelevant information that could complicate the analysis, or even lead to divergence.

Related Work. The most closely related technique is predicate abstraction [12]. This is implemented using the lazy paradigm in the BLAST model checker [7], and in a number of software model checkers [2,4,3] using a counterexample-based refinement loop. The advantage of the present method over predicate abstraction is that it avoids computing the abstract "post" operator. That is, in predicate abstraction, computing the set of successors of a set of abstract states requires an exponential number of calls to a decision procedure in the worst case. Because of this, weak approximations are typically used, such as the Cartesian or "Boolean Programs" approximations [1], with the associated need for refinement in case of failures. Even with approximations, computing the abstract post operator (or abstract transition relation) is still the dominant cost. By contrast, the present method requires just one call to a decision procedure for each error vertex reached, and one for each covering test.

The method is also closely related to the interpolation-based model checking method of [10]. That work only treated finite-state systems. In principle the method could be generalized to infinite-state programs, however it would require applying a decision procedure to an unfolding of the entire program up to some depth k. This would almost certainly be impractical. Using the lazy abstraction method, we only apply the decision procedure to individual program paths leading to error locations, greatly reducing the burden on the prover.

Outline of the Paper. In section 2, we will formalize the lazy interpolation-based model checking procedure, proving some results about soundness and termination. Then in section 3, we describe an implementation of the procedure in a software model checking tool called IMPACT, and compare the performance of this tool to the lazy predicate abstraction approach implemented in BLAST. Experiments using a small set of device driver benchmarks show a performance improvement of one to two orders of magnitude using the new method. Finally in section 4, we conclude and consider some future directions for research.

2 Lazy Interpolant-Based Model Checking

Throughout this paper, we will use standard first-order logic (FOL) and 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 use FOL 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 $\mathcal{L}(S \cup S')$.

For our purposes, a *program* is a tuple $(\Lambda, \Delta, l_i, l_f)$, where Λ is a finite set of program locations, Δ is a set of *actions*, $l_i \in \Lambda$ is the initial location and $l_f \in \Lambda$ is the error location. An *action* is a triple (l, T, m), where $l, m \in \Lambda$ are respectively the entry and exit locations of the action, and T is a transition formula. A *path* π of a program is a sequence of transitions of the form (l_0, T_0, l_1) $(l_1, T_1, l_2) \cdots (l_{n-1}, T_{n-1}, l_n)$. The path is an *error path* when $l_0 = l_i$ and $l_n = l_f$. The *unfolding* $\mathcal{U}(\pi)$ of path π is the sequence of formulas $T_0^{\langle 0 \rangle}, \ldots, T_n^{\langle n-1 \rangle}$, that is, the sequence of transition formulas $T_0 \ldots T_{n-1}$, with each T_i shifted i time units into the future.

We will say that path 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 is said to be *safe* when every error path of the program is infeasible. An *inductive invariant* of a program is a map $I : \Lambda \to \mathcal{L}(S)$, such that $I(l_i) \equiv \textsc{True}$ and for every action $(l, T, m) \in \Delta$, $I(l) \wedge T$ implies $I(m)'$. A *safety invariant* of a program is an inductive invariant such that $I(l_f) \equiv \textsc{False}$. Existence of a safety invariant of a program implies that the program is safe.

To simplify presentation of the algorithms, we will assume that every location has at least one outgoing action. This can be made true without affecting program safety by adding self-loops.

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
- $\hat{A} \in \mathcal{L}(A) \cap \mathcal{L}(B)$.

The Craig interpolation lemma [5] states that an interpolant always exists for inconsistent formulas in FOL. To handle 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 $1 \leq i < n$, $\hat{A}_i \in (\mathcal{L}(A_1 \ldots A_i) \cap \mathcal{L}(A_{i+1} \ldots A_n))$.

That is, the i-th element of the interpolant is a formula over the common vocabulary the prefix $A_0 \dots A_i$ and the suffix $A_{i+1} \dots A_n$, and each interpolant implies the next, with A_i. If Γ is quantifier-free, we can derive a quantifier-free interpolant for Γ from a refutation of Γ, in certain interpreted theories [11].

Program Unwindings. We now give a definition of a program unwinding, and an algorithm to construct a complete unwinding using interpolants. For two vertices v and w of a tree, we will write $w \sqsubset v$ when w is a proper ancestor of v.

Definition 1. *An* unwinding *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ *is a quadruple* (V, E, M_v, M_e)*, where* (V, E) *is a directed tree rooted at* ϵ*,* $M_v : V \to \Lambda$ *is the vertex map, and* $M_e : E \to \Delta$ *is the edge map, such that:*

- $M_v(\epsilon) = l_i$
- *for every non-leaf vertex* $v \in V$*, for every action* $(M_v(v), T, m) \in \Delta$*, there exists an edge* $(v, w) \in E$ *such that* $M_v(w) = m$ *and* $M_e(v, w) = T$*.*

Definition 2. *A* labeled unwinding *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ *is a triple* $(U, \psi, \triangleright)$*, where*

- $U = (V, E, M_v, M_e)$ *is an unwinding of* $\mathcal{A}$
- $\psi : V \to \mathcal{L}(S)$ *is called the vertex labeling, and*
- $\triangleright \subseteq V \times V$ *is called the covering relation.*

A vertex $v \in V$ *is said to be* covered *iff there exists* $(w, x) \in \triangleright$ *such that* $w \sqsubseteq v$*. The unwinding is said to be* safe *iff, for all* $v \in V$*,* $M_v(v) = l_f$ *implies* $\psi(v) \equiv$ FALSE*. It is* complete *iff every leaf* $v \in V$ *is covered.*

Definition 3. *A* labeled unwinding $(U, \psi, \triangleright)$ *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$*, where* $U = (V, E, M_v, M_e)$*, is said to be* well-labeled *iff:*

- $\psi(\epsilon) \equiv$ TRUE*, and*
- *for every edge* $(v, w) \in E$*,* $\psi(v) \wedge M_e(v, w)$ *implies* $\psi(w)'$*, and*
- *for all* $(v, w) \in \triangleright$*,* $\psi(v) \Rightarrow \psi(w)$*, and* w *is not covered.*

Notice that, if a vertex is covered, all its descendants are also covered. Moreover, we do not allow a covered vertex to cover another vertex. To see why, consider the unwinding of Figure 3. Here, vertex y covers x, but is itself covered, since its ancestor v is covered by w. This might seem acceptable, since any states reachable from y should be reachable from w through its descendant z. However, this is not the case. Because the vertex labels are approximate, it may be that $\psi(y) \not\Rightarrow \psi(z)$. Thus, z may not reach all states reachable from x.

Theorem 1. *If there exists a safe, complete, well-labeled unwinding of program* $\mathcal{A}$*, then* $\mathcal{A}$ *is safe.*

Proof. Let U be the set of uncovered vertices, and let function M map location l to $\bigvee\{\psi(v) \mid M_v(v) = l,\ v \in U\}$. M is a safety invariant for $\mathcal{A}$. □

We now describe a semi-algorithm for building a complete, safe, well-labeled unwinding of a program. The algorithm terminates if the program is unsafe,

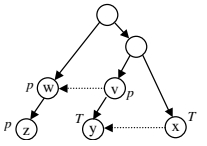


Fig. 3. Example showing why covered vertices must not cover others

global variables: V a set, $E \subseteq V \times V$, $\triangleright \subseteq V \times V$ and $\psi : V \to \mathit{wff}$

procedure EXPAND($v \in V$):
  if v is an uncovered leaf then
    for all actions $(M_v(v), T, m) \in \Delta$
      add a new vertex w to V and a new edge (v, w) to E;
      set $M_v(w) \leftarrow m$ and $\psi(w) \leftarrow$ TRUE;
      set $M_e(v, w) \leftarrow T$

procedure REFINE($v \in V$):
  if $M_v(v) = l_f$ and $\psi(v) \not\equiv$ FALSE then
    let $\pi = (v_0, T_0, v_1) \cdots (v_{n-1}, T_{n-1}, v_n)$ be the unique path from ϵ to v
    if $\mathcal{U}(\pi)$ has an interpolant $\hat{A}_0, \ldots, \hat{A}_n$ then
      for $i = 0 \ldots n$:
        let $\phi = \hat{A}_i^{\langle -i \rangle}$
        if $\psi(v_i) \not\models \phi$ then
          remove all pairs $(\cdot, v_i)$ from $\triangleright$
          set $\psi(v_i) \leftarrow \psi(v_i) \wedge \phi$
    else abort (program is unsafe)

procedure COVER($v, w \in V$):
  if v is uncovered and $M_v(v) = M_v(w)$ and $v \not\sqsubseteq w$ then
    if $\psi(v) \models \psi(w)$ then
      add (v, w) to $\triangleright$;
      delete all $(x, y) \in \triangleright$, s.t. $v \sqsubseteq y$;

Fig. 4. Three basic unwinding steps

but may not terminate if it is safe (which is expected, since program safety is undecidable). We first outline a non-deterministic procedure with three basic steps: EXPAND, which generates the successors of a leaf vertex, REFINE, which refines the labels along a path, labeling an error vertex FALSE, and COVER, which expands the covering relation. These steps are shown in Figure 4.

The interpolant in REFINE can be generated from a refutation of $\mathcal{U}(\pi)$, by the method of [11]. Each of the three steps preserves well-labeledness of the unwinding. In REFINE, the first two well-labeledness conditions are guaranteed by the properties of interpolants (*i.e.*, $\hat{A}_0 =$ TRUE and each interpolant formula implies the next). When we strengthen $\psi(v)$, we remove all arcs $(\cdot, v)$ in the covering relation, since a vertex covered by v may no longer be covered after strengthening v. In COVER, if a vertex v becomes covered, then all descendants of v are also covered. This means that any existing covering arcs (x, y) where $v \sqsubseteq y$ must be removed to maintain well-labeledness. If REFINE succeeds, then $\psi(v)$ must be FALSE (since $\hat{A}_n$ is always FALSE). Thus, to make the unwinding safe, we have only to apply REFINE to every error vertex. Finally, when none of the three steps can produce any change, the unwinding is both safe and complete, so we know the original program is safe.

```
procedure Close(v ∈ V):
    for all w ∈ V s.t. w ≺ v and M_v(w) = M_v(v):
        Cover(v, w)

recursive procedure DFS(v ∈ V):
    Close(v)
    if v is uncovered then
        if M_v(v) = l_f then
            Refine(v);
            for all w ⊑ v: Close(w)
        Expand(v);
        for all children w of v: DFS(w)

procedure Unwind:
    set V ← {ε}, E ← ∅, ψ(ε) ← True, ▷ ← ∅
    while there exists an uncovered leaf v ∈ V:
        for all w ∈ V s.t. w ⊏ v: Close(w);
        DFS(v)
```

Fig. 5. DFS unwinding strategy

To build a well-labeled unwinding, we now have only to choose a strategy for applying the three unwinding rules. The most difficult question is when to apply Cover. Covering one vertex can result in uncovering others. Thus, applying Cover non-deterministically may not terminate. To avoid this possibility, we define a total order $\prec$ on the vertices. This order must respect the ancestor relation. That is, if $v \sqsubset w$ then $v \prec w$. For example, we could define $\prec$ by a pre-order traversal of the tree, or by numbering the vertices in order of creation. We then restrict Cover to pairs (v, w) such that $w \prec v$. Now suppose that in adding a covering arc (v, w), we remove (x, y), where $v \sqsubseteq y$. Then by transitivity, we must have $v \prec x$. Thus, covering a vertex v can only result in uncovering vertices greater than v. This implies that we cannot apply Cover infinitely.

We will say that a vertex v is *closed* if either it is covered, or no arc (v, w) can be added to $\triangleright$ (while maintaining well-labeledness). The procedure Close of Figure 5 closes a vertex. We would like to guarantee that when a vertex is expanded, all of its ancestors are closed, thus we do not expand a vertex that could be covered instead. We could, of course, call Close on all the ancestors of a vertex v before expanding it. This would be costly, however. A more efficient strategy is shown in Figure 5. The procedure Unwind locates an uncovered leaf, then performs a local depth-first search around that leaf. During the search, it maintains the invariant that all ancestors of the currently visited leaf vertex v are closed. Moreover, all the vertices on the DFS stack are children of ancestors of v. Thus, when we pop a vertex off of the stack, we have only to call Close on the new vertex to re-establish the invariant. After calling Refine on an error vertex, the procedure calls Close on all of the ancestors v. This can be improved somewhat by only re-closing those vertices that were actually strengthened by Refine.

Theorem 2. *If procedure* Unwind *terminates without aborting on program $\mathcal{A}$, then $\mathcal{A}$ is safe.*

Proof. Since only the operations Expand, Refine and Cover alter the unwinding, and these preserve well-labeledness, the resulting unwinding is well-labeled.

Further, since all error vertices are refined, the unwinding is safe. Since the procedure terminates only when there are no uncovered leaves, the final unwinding is complete. Thus, by Theorem 1, program $\mathcal{A}$ is safe. □

Termination. Due to decidability considerations, we do not expect the unwinding to terminate in all cases. However, in the finite-state case, or in general when the language $\mathcal{L}(S)$ has bounded ascending chains, we can show termination. A finite ascending chain is a sequence of formulas $\phi_0, \phi_1, \ldots, \phi_n$ such that for all $0 \leq i < j \leq n$, $\phi_j \not\Rightarrow \phi_i$. We will say that a language L is k-bounded, for integer k, if all ascending chains in L have length at most k. For example, the Boolean formulas over n variables are $2^n + 1$-bounded.

Theorem 3. *If $\mathcal{L}(S)$ is k-bounded, then procedure* Unwind *terminates or aborts.*

Proof. Procedure DFS maintains the invariant that all ancestors of v are closed. Thus, there are no $x \sqsubset w \sqsubseteq v$ such that $M_v(x) = M_v(w)$ and $\psi(w) \Rightarrow \psi(x)$ (else w would not be closed). Thus, for any location l, the formulas $\phi(w)$ where $M_v(w) = l$ and $w \sqsubseteq v$ form an ascending chain. Since $\mathcal{L}(S)$ is k-bounded, it follows that the path from ϵ to v contains at most $|\Lambda| \cdot k$ vertices. Thus the depth of the tree is bounded. As argued above, Cover cannot continue to cover vertices infinitely. Thus, in the main loop, always eventually Close fails to cover a new vertex, or the loop terminates. In the former case, vertex v remains uncovered, and is thus expanded in procedure DFS. However, we cannot expand vertices infinitely, since the tree depth is bounded. Thus, the loop must terminate (or abort in Refine). □

A Weak Notion of Completeness. In general, the FO formulas over a given vocabulary S have infinite ascending chains. Thus, the above termination result is not generally applicable. However, by restricting the language of the interpolants, we can force termination (perhaps without deciding safety). That is, given a language L, an L-restricted interpolant for a sequence Γ is an interpolant for Γ in which all formulas are contained in L. Techniques for computing L-restricted interpolants are described in [9]. Given a language L, let us define an unwinding procedure Unwind(L) that differs from Unwind only in that "interpolant" in procedure Refine is replaced by "L-restricted interpolant". If language L is k-bounded, then Unwind(L) must terminate or abort. Moreover, in [9] it is shown that if program $\mathcal{A}$ has an inductive invariant expressible in L, then every error path of $\mathcal{A}$ has an L-restricted interpolant. Thus Unwind(L) cannot abort, and must terminate proving safety.

We can use this idea to create a procedure that is complete in the limited sense that it eventually verifies all programs that have inductive invariants expressible as quantifier-free formulas in a suitable FO theory. That is, we define an infinite chain of k-bounded, quantifier-free restriction languages $L_0 \subseteq L_1 \cdots$, such that every formula is contained in some L_k.[1] If a program has a quanti-

[1] Quantifier-freeness is required so that the entailment tests in Refine and Close are decidable. Otherwise completeness is relative to an oracle for the theory.

```
procedure ForceCover(v, w ∈ V)
    let x be the nearest common ancestor of v and w
    let π = (v_0, T_0, v_1) ··· (v_{n-1}, T_{n-1}, v_n) be the unique path from x to v
    let Γ = ψ(x) · U(π) · ¬ψ(w)^⟨n⟩
    if Γ has an interpolant Â_0, ..., Â_{n+2} then
        for i = 0 ... n:
            let φ = Â_{i+1}^⟨−i⟩
            if ψ(v_i) ⊭ φ then
                remove all pairs (·, v_i) from ▷
                set ψ(v_i) ← ψ(v_i) ∧ φ
```

Fig. 6. Procedure to force covering of one vertex by another

fier free safety invariant in the theory, then it has an invariant in some L_k. We start with L_0 and each time Unwind(L_i) aborts, we move on to L_{i+1}. When we reach L_k, the Unwind(L_k) must terminate. Thus, our approach is complete in the limited sense that it verifies (eventually) any program with a quantifier-free safety invariant in the theory (this is precisely the set of programs that we can verify with predicate abstraction *if* we can guess the right atomic predicates). Of course, in practice we must choose the restriction languages L_k carefully, so that termination occurs for a small value of k.

Forced Covering. To speed convergence of the unwinding procedure, we can use interpolant-based refinement to force a vertex v to be covered by some other vertex w. We will call this a *forced covering*. Suppose that v and w have nearest common ancestor x in the unwinding. We construct the characteristic formula for the path from x to v, asserting $\psi(x)$ at the beginning, and $\neg\psi(w)$ at the end. If this is infeasible (meaning $\psi(w)$ must hold at v) we strengthen all the vertices on the path from x to v by the corresponding interpolant formulas. Thus, we ensure that w covers v. This procedure is depicted in Figure 6. Clearly, attempting all possible forced coverings could be costly. In practice, before expanding a vertex we attempt a forced covering by a few recently generated vertices representing the same program location. This substantially reduces the part of the unwinding that we must explore.

Other Optimizations. As in other work using interpolants [6,8], we generate the characteristic formula of a path in static single-assignment (SSA) form. That is, we create a new instance of a program variable only when that variable is modified. This eliminates a large number of constraints of the form $x^{\langle i+1\rangle} = x^{\langle i\rangle}$ that occur when a variable is unmodified by a program statement.When refining a program path, we also use a simple slicing (or "cone-of-influence" reduction) to remove from the program path any assignments that cannot affect the feasibility of the path. Slicing typically removes a large fraction of the assignments in the path, especially initializations of global variables that are not referenced. It should be noted, however, that slicing can affect completeness, since it is possible that a variable that is not referenced is nonetheless necessary to express an inductive invariant (it might even be an auxiliary variable added by the user for this purpose). In practice, however, this has not been observed to occur, and slicing yields a substantial performance improvement.

Finally, in the REFINE and COVER steps, we must test whether one formula entails another, using a decision procedure. Since the same test tends to occur many times, it pays to memoize the decision procedure calls.

3 Experiments

The lazy interpolation-based unwinding procedure is implemented in a software model checking tool called IMPACT[2] (carrying on the tradition of violent acronyms for software model checkers). In this section, we compare the interpolant-based method of IMPACT with the predicate abstraction approach of BLAST. The benchmarks we use are device drivers from the Microsoft Windows DDK, written in C. They were used as test cases in [6]. Each driver is provided with a test harness (*i.e.*, a main program that calls the driver functions appropriately in a non-deterministic manner) and is instrumented with auxiliary variables and safety assertions that test whether whether certain rules are obeyed in calling the kernel API functions.[3] All six of the example programs are safe. To check the implementation of IMPACT, however, we inserted three errors into each example program. IMPACT detected all 18 errors, each in at most a few seconds. Performance data are reported only for the safe versions.

IMPACT is based on the interpolating prover of [9]. This prover supports a first-order theory with equality, uninterpreted function symbols, and integer difference-bound arithmetic (*i.e.*, predicates of the form $x - y \leq c$, $x \leq c$ or $x \geq c$, where c is a constant). It also supports first-order arrays, with interpreted "select" and "store" functions. Support for full linear arithmetic is also possible, but currently not for integer models.

To handle C programs, we first reduce them to Simple Goto Programs (SGP's). These are programs containing only conditional goto statements, assignments and assertions, and whose only data types are unbounded integers and arrays of unbounded integers. Pointers and records are eliminated by this translation, and function calls are in-lined. This reduction was done using a modified version of the SATABS infrastructure [4]. Unfortunately, space does not permit a description of the translation process here.

Once a C program has been translated to a simple goto program, we can model it formally in the logic of the prover. The logic contains operations on arrays, as well as limited arithmetic. We model the unsupported integer operations (such as the bit-wise operators) with uninterpreted functions (thus we may fail to prove safety if it depends on properties of these operators). An assertion in the program is modeled by a conditional branch to the error state l_f. Transitions in the model correspond to basic blocks in the goto program. Having modeled the program, we can then verify safety using procedure UNWIND(L), where L is the restriction language for interpolation. We use the same sequence of restriction languages L_k as in [9]. This restricts the constants in arithmetic formulas to fall in a certain finite set that depends on k, and also restricts depth of function

[2] Interpolating software Model checker without Predicate abstrACTion.
[3] Benchmarks available from the author.

Table 1. Performance statistics on device driver benchmarks

name	source loc	SGP loc	BLAST time(s)	IMPACT time(s)	speedup	BLAST preds	BLAST post(s)	IMPACT interp(s)	BLAST vtcs	IMPACT vtcs
kbfiltr	12K	2.3K	26.3	3.15	8.3	25	23.3	2.2	1651	744
diskperf	14K	3.9K	102	20.0	5.1	84	92.2	19.3	3232	3885
cdaudio	44k	6.3K	310	19.1	16.2	108	265	11.9	5253	3257
floppy	18K	8.7K	455	17.8	25.6	105	404	16.9	9573	2518
parclass	138K	8.8K	5511	26.2	210	162	5302	22.9	8612	3720
parport	61K	13K	8084	37.1	218	224	7965	31.0	63.5K	12.7K

symbol nesting as a function of k. In fact, all of the example programs can be verified with restriction language L_0.

For comparison to predicate abstraction approach, we use the BLAST software model checker [7]. This tool is in some ways a good comparison, since it is also based on the "lazy abstraction" paradigm (using predicate abstraction instead of interpolation to refine paths). In addition, it uses the same interpolating prover to generate atomic predicates that IMPACT uses for path refinement. Thus in principle both tools should be able to construct the same class of safety invariants. On the other hand, the implementations are independent, so observed performance differences may be due in part to implementation efficiencies. In principle the closest comparison could be obtained by running both programs on the same SGP. However, as it turns out the performance of BLAST was significantly better when run on the original C source code. This may be because the elimination of pointers prevented the use of some pointer-based optimizations in BLAST. For this reason, we present performance numbers for BLAST as run on the original source code. We use the standard BLAST option that assigns to each new vertex all of the predicates that have been used for program locations in the same function scope. This tends to increase the number of predicates at each vertex, but reduces the number of refinements needed, thus yielding better performance.[4]

Table 1 compares the run time performance of BLAST and IMPACT on the six device driver examples. The first three columns show the name of the example, the number of textual lines in the source code, and the number of lines in the SGP. The last probably provides a better representation of the code size, since the source code contains much white space and many redundant declarations. The next two columns provide the run times for BLAST and IMPACT. Both are run on a 3GHz Intel Xeon processor. These times represent only the model checking process, and do not include time for parsing or translation to an SGP. The next column shows the speedup of IMPACT relative to BLAST. For the small examples, IMPACT has about an order of magnitude advantage, which increases to two orders of magnitude for the large examples.

The explanation for the performance difference may lie in the fact that the abstract post computation becomes increasingly expensive as the programs get

[4] The BLAST options used were `-msvc -nofp -dfs -tproj -cldepth 1 -predH 6 -scope -nolattice -clock`.

Table 2. Performance statistics for revised BLAST

name	source loc	SGP loc	BLAST time(s)	IMPACT time(s)	speedup	BLAST preds	BLAST post(s)	IMPACT interp(s)	BLAST vtcs	IMPACT vtcs
kbfiltr	12K	2.3K	11.9	3.15	3.8	38	6.6	2.2	1009	744
diskperf	14K	3.9K	117	20.0	5.9	119	49.8	19.3	1855	3885
cdaudio	44k	6.3K	202	19.1	10.6	180	114	11.9	3400	3257
floppy	18K	8.7K	164	17.8	9.2	154	77.9	16.9	2856	2518
parclass	138K	8.8K	463	26.2	17.7	242	175	22.9	5003	3720
parport	61K	13K	324	37.1	8.7	280	156	31.0	10.4K	12.7K

larger and the number of predicates increases. The table shows some run-time statistics that bear this out. Columns 7–9 show the number of atomic predicates used by BLAST, the amount of time spent by BLAST in the predicate image computation, and the amount of time spent by IMPACT in computing interpolants for path refinement. It is clear that avoiding the predicate image computation provides a significant advantage. The last two columns of the table show the number of vertices in the final unwinding for both BLAST and IMPACT. BLAST expands more vertices (though not enough to fully account for the performance difference). This may be because the predicate images computed by BLAST are stronger than necessary. Thus BLAST distinguishes states that need not be distinguished, resulting in a larger unwinding.

After this paper was originally submitted, Ranjit Jhala improved the performance of BLAST by making it less "lazy". In this version, each new vertex in the unwinding is assigned all the predicates seen thus far for the same program location, or if there are none, then predicates of its parent. This slightly "eager" approach greatly reduces the number of refinement steps. The reduction in refienements makes it practical to use only the predicates from the same *location*, rather than the same function scope, which reduces the number of predicates per vertex and thus speeds the predicate image computation substantially. Table 2 shows comparison data for this new version.[5] The performance gap between BLAST and IMPACT is now considerably smaller (only one order of magnitude). It could be that computing some state information in an eager manner would reduce the number of refinement steps of IMPACT as well. We leave this question for future research.

4 Conclusion

We have described a method that uses interpolation rather than predicate abstraction in the lazy abstraction paradigm. This avoids the most costly operation of predicate abstraction, the abstract image computation. In contrast to the interpolation-based model checking method of [10], it avoids constructing and refuting an unfolding of the entire program. Instead, the interpolating prover is

[5] BLAST options for this experiment were `-msvc -nofp -craig 2 -scope -cldepth 1 -bfs` except for cdaudio, which also required `-clock`. No single set of options was able to verify all the examples.

applied only to individual program paths, greatly lessening the burden on the prover. This makes it possible to apply the interpolation-based approach to the verification of infinite-state sequential programs. For a small collection of device driver examples, a run-time improvement of one to two orders of magnitude was obtained, relative to the lazy predicate abstraction approach. Although a greater variety of examples is clearly needed to study the trade-offs between the two methods, the experiments show that the interpolation method has the potential to provide a substantial performance improvement.

There are several potentially interesting topics for future research. Consider, for example, the following simple C program fragment:

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

A safety invariant of this program requires a universal quantifier over the index of the array. Thus, predicate abstraction methods that use atomic predicates cannot verify this program. However, in [11] it is shown that an interpolating prover can be used to generate interpolants with quantifiers. This opens the possibility of generating quantified inductive invariants with the present method. There are several challenges involved in this. First the decision procedure must handle quantified formulas. Since the validity of quantified formulas is undecidable, we must have heuristics to instantiate quantifiers. Second, we must somehow prevent the number of quantifiers in the interpolants from increasing without bound. Although these problems remain to be solved, using a naïve approach to quantifier instantiation it is possible to verify simple programs like the above. Thus, it may be possible to use the method to verify properties that depend, for example, on the contents of arrays.

It also seems possible that the interpolation approach can be made to scale better by using function summaries, in an approach that might be called "summaries on demand". If we refute a program path that contains the expansion of a procedure call, we can derive an interpolant that is an over-approximation of the transition relation of the procedure (in the same way that transition relation approximations are derived in [8]). This approximation can be used as an abstraction (summary) of the procedure. When an error path is found not to be refutable, it might be refined by expanding one or more summarized functions, which would strengthen the summaries of the expanded functions. Thus, there seems to be scope for both enriching the class of properties that can be verified, and for improving the performance of the method on large programs.

Acknowledgments. Thanks to Daniel Kröning for providing the SATABS tool infrastructure used in this work, to Ranjit Jhala for help with Blast, and to the anonymous reviewers for useful comments and corrections.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.

2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
3. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.
4. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer Verlag, 2005.
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. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
8. R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *LNCS*, pages 39–51. Springer, 2005.
9. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.
10. K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, pages 1–13, 2003.
11. Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
12. H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.