Automatically Refining Abstract Interpretations

Bhargav S. Gulavani* Supratik Chakraborty* Aditya V. Nori† and Sriram K. Rajamani†

*IIT Bombay †Microsoft Research India

Abstract. Abstract interpretation techniques prove properties of programs by computing abstract fixpoints. All such analyses suffer from the possibility of false errors. We present three techniques to automatically refine such abstract interpretations to reduce false errors: (1) a new operator called *interpolated widen*, which automatically recovers precision lost due to widen, (2) a new way to handle disjunctions that arise due to refinement, and (3) a new refinement algorithm, which refines abstract interpretations that use the join operator to merge abstract states at join points. We have implemented our techniques in a tool Dagger. Our experimental results show our techniques are effective and that their combination is even more effective than any one of them in isolation. We also show that Dagger is able to prove properties of C programs that are beyond current abstraction-refinement tools, such as Slam [4], Blast [14], Armc [18], and our earlier tool [11].

1 Introduction

Abstract interpretation [7] is a general technique to compute sound fixpoints for programs. Such fixpoint computations have to lose precision in order to guarantee termination. However, precision losses can lead to false errors. Over the past few years, counterexample driven refinement has been successfully used to automatically refine *predicate abstractions* (a special kind of abstract interpretation) to reduce false errors [4, 14, 18]. This has spurred significant research in counterexample guided discovery of "relevant" predicates [5, 8, 13, 16, 19]. A natural question to ask therefore is whether counterexample guided automatic refinement can be applied to any abstract interpretation. A first attempt in this direction was made in [11] where widen was refined by convex hull in the polyhedra domain. This was subsequently improved upon in [21] where widen was refined using extrapolation. This paper improves the earlier efforts in three significant ways that combine to give enhanced accuracy and efficiency. First, we propose an interpolated widen operator that refines widen using interpolants. Second, we propose a new algorithm to implicitly handle disjunctions that occur during refinement. Finally, we propose a new algorithm to refine abstract interpretations that use the join operator to merge abstract states at program locations where conditional branches merge. We have built a tool Dagger that implements these ideas. Our empirical results show that Dagger outperforms a number of available tools on a range of benchmarks, and is able to prove array-bounds properties of several programs that are beyond the reach of current abstraction-refinement tools such as Slam [4], Blast [14], Armc [18], and our earlier tool [11].

```
 0: int x=0; y=0; z=0; w=0;

 1: while (*) {
 2:  if (*)
 3:   {x = x+1; y = y+100;}
 4:  else if (*) {
 5:   if (x >= 4)
 6:    {x = x+1; y = y+1;}
     }
---------------------------------------
|7:  else if (y > 10*w && z >= 100*x) |
|8:    {y = -y;}                      |
|9:  w = w+1; z = z+10;               |
---------------------------------------
    }
10: if (x >= 4 && y <= 2)
11:   error();
```

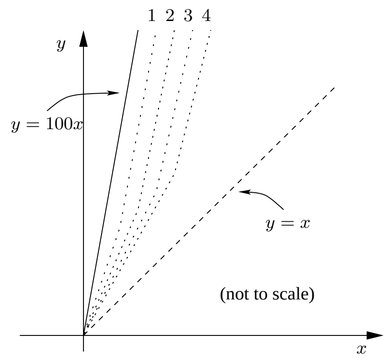

Fig. 1. Example program

The widen operator is typically used in abstract interpreters to generate invariants by generalizing from multiple symbolic executions. However, widen is unaware of the target property that needs to be verified, and may result in approximations too coarse to prove the property. Interpolants offer a complementary generalization capability by providing succinct reasons for spuriousness of counterexamples. However, interpolants are generated with respect to specific counterexample traces, and are not guaranteed to be fixpoints with respect to all executions. By combining the strengths of interpolants and widen in an effective way, *interpolated widen* gives benefits of both.

To illustrate the benefit of using interpolants in conjunction with invariants obtained by widening, consider the program in Figure 1 (ignore the boxed code for now). The error at line `11` is unreachable. The inductive loop invariant $x \leq y \leq 100x$ suffices to prove unreachability of the error. However, if we refine widen using convex hull as in [11], we obtain the weaker invariant $100x \geq y$ that does not help in proving the program correct. The polyhedra obtained after the ith such refinement iteration is indicated in Figure 1 by the region between the line $y = 100x$ and the ith dotted boundary. This dotted boundary is then discarded in subsequent widen operations, and the abstract fixpoint intersects the error. Therefore the refinement of widen to convex hull continues ad infinitum, giving the imprecise invariant $100x \geq y$. Note that the extrapolation technique for polyhedra given in [21] will also face a similar problem. In contrast, an interpolant generation technique [16] can compute the interpolant $y \geq x$ easily by analyzing a counterexample. By using this in conjunction with the invariant obtained by widen, we obtain the stronger invariant $x \leq y \leq 100x$, which is strong enough to prove that the error at line `11` is unreachable.

In the above example, $y \geq x$ is itself a strong enough loop invariant to prove unreachability of the error. It may therefore appear that interpolation based techniques perform better than widen based techniques. However, interpolation alone does not work in all cases. To illustrate this, consider the same example including the boxed code (lines `7-9`). In this program line `8` is unreachable. The inductive invariant required for proving the error unreachable is now $(x \leq y \leq 100x) \wedge (z = 10w)$. There is no obvious reason why interpolation techniques like [16,19] will choose $100x \geq y$ as part of an interpolant among the many possible interpolants during counterexample analysis. Experiments show that ARMC, which uses a sophisticated interpolation algorithm [19], does not terminate on this example in 2000s. STING [20] does not generate invariants strong enough to prove the error unreachable either. The refinement engine of BLAST is equipped to generate only difference and bounds predicates, hence it fails to prove the program correct. Since the coefficient 100 in the required predicate $100x \geq y$ is large, recursively enumerating interpolants, as suggested in [16], is also unlikely to work well. Widen based techniques like [11] can easily generate invariants like $100x \geq y$ and $z = 10w$, but not $x \leq y$, which can be easily generated by interpolant based techniques. Thus, by combining invariants obtained by widen with interpolants obtained during counterexample analysis, we obtain the right inductive invariant needed to prove the property. Further, our empirical results (see Section 4) show that interpolated widen is better than superficially combining widen and interpolation, i.e., by first computing invariants using widen, and then using them to strengthen the transition relation in interpolation based predicate abstraction frameworks, as suggested in [15].

Refining widen using specific operations in polyhedral abstract domains has been used earlier [11, 21]. While the widen up-to operator of [21] does not guarantee elimination of spurious counterexamples, the intuition behind widen up-to and extrapolation are useful. In [12] Halbwachs et al. introduced the widen up-to operator to improve the precision of widen with pre-computed (static) thresholds. The use of dynamically computed interpolants to refine widen is an original contribution of our work, and can be viewed as a generalization of the widen up-to operators of [12, 21]. Since interpolants provide succinct reasons for spuriousness of counterexamples (by referring only to common variables between a pair of formulas being interpolated), we enjoy the benefits of ideas in [11,21] while potentially using simpler/fewer predicates. Unlike [11, 21], we can also leverage independent advances in widening [3, 10] and interpolation techniques [5, 16, 19] in a simple framework. In [15], predicate abstraction based analysis is improved by using weak invariants discovered by widen in an initial pass. However, potentially stronger invariants that may be discovered by widen after few iterations of refinement are not considered. In contrast, our analysis based on interpolated widen can benefit from such stronger invariants discovered later, especially if sophisticated widening techniques [2, 3, 10] are used.

In addition to widen, if the join operator in an abstract domain loses precision, we need disjunctions to recover precision losses that are necessary to prove a property. However, this makes us work over a powerset domain, where operations like interpolation and widen are expensive. We propose a technique that implicitly uses disjunctions to recover precision as appropriate, while ensuring that interpolation and widen are applied only on base abstract domain (and not powerset domain) elements. This contrasts with other approaches [11, 21] that use similar base abstract domains but must use powerset widening.

In programs with conditional branches, the tree-based exploration used in our earlier work [11] can result in traversing an exponential (in size of program) number of paths. This can be avoided in abstract interpretation by using join operations when different branches of conditional statements merge, in addition to performing widen operations at loopheads. This is indeed a DAG-based exploration. Therefore, an interesting question to ask is: *can we perform counterexample driven automatic refinement with a* DAG*-based exploration*? In this paper, we propose a refinement algorithm that achieves this and also gives progress guarantees. Counterexample-DAG based predicate abstraction has been used earlier for programs with finite domain variables [8]. In contrast, our DAG based refinement is used to refine imprecisions that arise due to the join operator at merge nodes. In [9], Fischer et. al. used predicated dataflow lattices to improve precision lost by join operation. However, the dataflow lattices considered in [9] are of finite height and hence do not require a widen operator.

Our approach, like those of [11, 15, 21], benefits from cheap image/preimage operations of abstract domains like octagons and polyhedra, as opposed to expensive image/preimage computations in predicate abstraction. In [17], McMillan showed how abstract exploration for predicate abstraction can be performed by way of computing interpolants, thus eliminating the need for the expensive image computation. While this is a powerful technique, it does not benefit from predicates that can be easily discovered as invariants by widen but are more difficult to obtain as interpolants, and that are also crucial for proving a property. Beyer et al [5] introduced path programs to help discover such relevant predicates. If we view this as an advanced interpolation technique, our approach, like other predicate abstraction techniques, can only benefit from the predicates thus computed. Interestingly, our abstraction refinement algorithm can also be used to compute relevant predicates by analyzing path programs.

The remainder of the paper is organized as follows. In Section 2 we present the interpolated widen operation and discuss implicit handling of disjuncts. Section 3 discusses DAG-based refinements. Section 4 presents and analyzes experimental results from our tool Dagger, and Section 5 concludes the paper.

2 Refinement: interpolated widen and implicit disjuncts

Let V be a finite set of variables. A state s is a valuation to all variables in V. Let Σ be the (possibly infinite) set of all possible states. A program P_V over a set of variables V is a six-tuple $(L, E, R, l_0, \mathsf{Image}^\flat)$, where (i) L is a finite set of control locations in the program, representing possible valuations to the program counter, (ii) $E \subseteq L \times L$ is a set of control flow edges, (iii) $R \subseteq L$ is a set of error locations, (iv) l_0 is the initial program location, which is not in R, and cannot be the target of any control flow edge, and (vi) $\mathsf{Image}^\flat$ is a function from $2^\Sigma \times L \times L$ to 2^Σ, where $\mathsf{Image}^\flat(\sigma, l, l')$ is the set of states obtained by starting at some state in the set of states σ and executing the statements along the control flow edge (l, l'). The preimage operation $\mathsf{Preimage}^\flat$ is defined as the inverse of the image operation. We overload the $\mathsf{Image}^\flat$ and $\mathsf{Preimage}^\flat$ operations to operate over a sequence of edges in the obvious way.

We assume that the control flow graphs of our programs are connected reducible graphs, and that every location has at most two incoming control flow edges. An edge $e \in E$ is a *backedge* if it closes a cycle during a depth first traversal of the graph $\langle L, E \rangle$, starting at location l_0. A location l is called a *merge* location if it has two incoming edges, neither of which is a backedge. A location l is called a *loophead* if it has two incoming edges, and exactly one is a backedge.

A control location l is said to be reachable if there exists a path $(l_0, l_1, \ldots, l_n, l)$ in the control flow graph and a state $\sigma_0 \in \Sigma$, such that $\mathsf{Image}^\flat(\{\sigma_0\}, (l_0, l_1, \ldots, l_n, l))$ is not $\emptyset$. Our goal is to check if any error location $l_e \in R$ is reachable. A true counterexample is a sequence of control flow locations $(l_0, l_1, \ldots, l_n, l_e)$ such that $l_e \in R$ and there exists $\sigma_0 \in \Sigma$ satisfying $\mathsf{Image}^\flat(\{\sigma_0\}, (l_0, l_1, \ldots, l_n, l_e)) \neq \emptyset$. The counterexample is called spurious if $\mathsf{Image}^\flat(\{\sigma_0\}, l_0, l_1, \ldots, l_n, l_e) = \emptyset$ for every $\sigma_0 \in \Sigma$. The length of a counterexample $(l_0, l_1, \ldots, l_n, l_e)$ is one less than the length of the sequence $(l_0, l_1, \ldots, l_n, l_e)$.

Following [7], we use abstract interpretation of a program P_V over an abstract domain $\langle \Sigma^\sharp, \sqsubseteq, \top, \bot, \sqcup, \sqcap \rangle$, which is a complete lattice. In particular, we consider abstract domains where elements of $\Sigma^\sharp$ are formulas over V in a fragment of first order logic closed under Craig interpolation [13]. Every formula represents a set of states. The abstract image operation $\mathsf{Image}(s, l, l')$ takes an abstract element s and the control flow edge (l, l') and returns the abstract element s' obtained by abstractly executing the statements along the control flow edge (l, l'). The abstract $\mathsf{Preimage}$ operation is analogously defined. In the following discussion, we will assume that the Image and $\mathsf{Preimage}$ operations are exact. The effects of overapproximating Image and $\mathsf{Preimage}$ operations are briefly discussed later.

Widen. The widen [7] operator $\nabla : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$ is a binary operator such that for all $A, B \in \Sigma^\sharp$, we have (i) $A \sqsubseteq A \nabla B$, (ii) $B \sqsubseteq A \nabla B$, and (iii) for any strictly increasing sequence $A_0 \sqsubset A_1 \sqsubset \ldots$, if we define $B_0 = A_0$, $B_1 = B_0 \nabla A_1$, $B_2 = B_1 \nabla A_2, \ldots$, then there exists $i \geq 0$ such that $B_j = B_i$ for all $j > i$.

The *bounded widen* or *widen up-to* [12] operator with respect to a set T of abstract elements, $\nabla_T : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$, is a widen operator such that for any $C \in T$, if $A \sqsubseteq C$ and $B \sqsubseteq C$ then $A \nabla_T B \sqsubseteq C$.

Interpolant. For any two elements $A, E \in \Sigma^\sharp$ such that $A \sqcap E = \bot$, $I \in \Sigma^\sharp$ is said to be an interpolant [13] of A and E if (i) $A \sqsubseteq I$, (ii) $I \sqcap E = \bot$, and (ii) the formula representing I has only those variables that are common to the formulae representing A and E.

The widen operator $A \nabla B$ is used to guarantee termination when computing an abstract fixpoint. However, the imprecision introduced by widen may lead to false errors. This can happen if, for an abstract error state E, $(A \sqcup B) \sqcap E = \bot$, but $(A \nabla B) \sqcap E \neq \bot$. In this case, we propose to pick an interpolant I of $A \sqcup B$ and E, and use a bounded widen operator $A \nabla_{\{I\}} B$ to compute the abstract fixpoint. Such a bounded widen operator that uses interpolants as bounds is called *interpolated widen*. A primary insight of this paper is that if the parameter T of a bounded widen operator contains an interpolant, then a false error that occurs due to the imprecision of widen can be avoided.

Lemma 1. *Let $A, B, E \in \Sigma^\sharp$ be such that $(A \sqcup B) \sqcap E = \bot$. Let $I \in \Sigma^\sharp$ be an interpolant of $(A \sqcup B)$ and E, and let $T \subseteq \Sigma^\sharp$ be any set such that $I \in T$. Then $(A \sqcup B) \sqsubseteq (A \nabla_T B) \sqsubseteq I$ and $(A \nabla_T B) \sqcap E = \bot$.*

In the polyhedra abstract domain, bounding widen with constraints from the convex hull has been used in earlier work [11,21]. Although such constraints sep-

/* **Global:** program P_V, interpolant set T, abstract computation tree */

1. AbstractTREE

$n_0 \leftarrow \langle l_0, \top \rangle$; $T \leftarrow \emptyset$; $i \leftarrow 0$;
loop
  for all $n = \langle l, s \rangle$ such that $\mathsf{Depth}(n) = i$
    for all edges (l, l') in cfg
      $img \leftarrow \mathsf{Image}(s, l, l')$
      if $\neg\mathsf{Covered}(l', img)$
        if l' is loophead
          $s'' \leftarrow \mathsf{Sel}(l', n)$
          $img \leftarrow s'' \nabla_T (s'' \sqcup img)$
        Add $\langle l', img \rangle$ as child of n
  if $\exists n_e = \langle l_e, s_e \rangle$ such that $l_e \in R$
    $i \leftarrow \mathsf{RefineTREE}(n_e)$
  else if $\neg\exists$ node at depth $i + 1$
    "program correct"; exit
  else
    $i \leftarrow i + 1$
end loop

2. RefineTREE (n_e)

$\psi \leftarrow \top$; $curr \leftarrow n_e$; $i \leftarrow \mathsf{Depth}(n_e)$
while $i > 0$
  Let $\langle l', s' \rangle = curr$ and $\langle l, s \rangle = \mathsf{Parent}(curr)$
  if $\mathsf{Image}(s, l, l') \sqcap \psi = \bot$
    /* $curr$ is a refinement node */
    $s' \leftarrow \mathsf{ApplyRefinement}(l', \langle l, s \rangle, \psi)$
    $\mathsf{DeleteDescendents}(curr)$
    return $i - 1$
  $\psi \leftarrow \mathsf{Preimage}(\psi, l, l')$
  $curr \leftarrow \mathsf{Parent}(curr)$; $i \leftarrow i - 1$
end while
"program incorrect"; exit

3. ApplyRefinement $(l', \langle l, s \rangle, \psi)$

Let $n = \langle l, s \rangle$; $s'' \leftarrow \mathsf{Sel}(S_{l',n})$;
$img \leftarrow \mathsf{Image}(s, l, l')$
if $(s'' \sqcup img) \sqcap \psi = \bot$ **then**
  $T \leftarrow T \cup \mathsf{Interpolate}(s'' \sqcup img, \psi)$
  return $s'' \nabla_T (s'' \sqcup img)$
else
  return img

Fig. 2. Refinement using Interpolated Widen

arate the forward reachable states from the error, they may not be interpolants. Since interpolants often give succinct reasons for eliminating counterexamples, we choose to use them to bound widen.

For several abstract domains like polyhedra and octagons, the $\sqcup$ operation is inexact, in addition to having inexact ∇. Powerset extensions of these domains however have exact $\sqcup$. A primary drawback of powerset domains is the increased complexity of interpolation and other abstract domain operations. We therefore propose ways to avoid these operations on powerset domains, while using base abstract domains with inexact $\sqcup$ operator. The abstraction refinement algorithm using interpolated widen and tree based exploration is shown in Figure 2.

During the abstract fixpoint computation, procedure AbstractTREE stores the intermediate states as a tree (N, A, n_0) where N is the set of nodes, A is the set of edges and n_0 is the root node. Each node in N represents an abstract state during the computation, stored as a pair $\langle l, d \rangle \in L \times \Sigma^\sharp$. The tree thus constructed gives a model to perform counterexample driven refinement whenever an error location is reached during the abstract computation.

Let the function $\mathsf{Parent} : N \setminus \{n_0\} \to N$ give for each node $n \in N \setminus \{n_0\}$ its unique parent n' in the tree. Let the function $\mathsf{Depth} : N \to \mathbb{N}$ give for each node n the number of edges along the path from n_0 to n. Let $\mathsf{Covered}(l, s)$ be a predicate that returns True iff either $s = \bot$ or there exists a node $n = \langle l, s' \rangle$ in the tree such that $s \sqsubseteq s'$. Let $S_{l,n}$ denote the set of maximal abstract states among the abstract states at the predecessors of node n with location l. Note that there can be more than one node with maximal abstract state among the predecessors of node n with location l because of refinements as explained later. Given a set of abstract states S, the function $\mathsf{Sel}(S)$ deterministically returns one element from S (using heuristics mentioned in [11]).

Since every node in the abstract tree stores a single abstract state, the image computation in each step of forward exploration along a path in the tree gives a single abstract state. Therefore, when we reach loopheads, we need to widen a set

S of abstract states with the set $S \cup \{s\}$, where s is the newly computed abstract state. Given this special requirement, we define an operator $\nabla_T^p : \wp(\Sigma^\sharp) \times \Sigma^\sharp \to \wp(\Sigma^\sharp)$ that takes a set $S \subseteq \Sigma^\sharp$ and an element $s \in \Sigma^\sharp$, and returns the set of maximal elements in $S \cup \{\mathsf{Sel}(S)\, \nabla_T\, (\mathsf{Sel}(S) \sqcup s)\}$.

Lemma 2. *Let $S_0 \subseteq \Sigma^\sharp$ be a finite set. Consider the sequence $S_1 = S_0\, \nabla_T^p\, s_0$, $S_2 = S_1\, \nabla_T^p\, s_1, \ldots$, where $s_i \in \Sigma^\sharp$ for all $i \geq 0$. There exists $u \geq 0$ such that $S_v = S_u$ for all $v \geq u$.*

The computation of $\mathsf{AbstractTREE}$ starts with an abstract tree having a single node $n_0 = \langle l_0, \top \rangle$. Consider a node $n = \langle l, s \rangle$ and control flow edge (l, l'). If $\mathsf{Covered}(l', \mathsf{Image}(s, l, l'))$ returns False, a new node $n' = \langle l', s' \rangle$ is added as a child of n in the tree; otherwise a new node is not added. If (l, l') is not a backedge then s' is obtained as $\mathsf{Image}(s, l, l')$. Otherwise s' is computed using an interpolated widen operation as $s'' \nabla_T (s'' \sqcup \mathsf{Image}(s, l, l'))$, where $s'' = \mathsf{Sel}(S_{l',n})$. In computing s', we must also ensure that the invariant computation for the current path eventually terminates if no refinements are done in between. Lemma 2 gives this guarantee. If a node $n_e = \langle l_e, s_e \rangle$ with $l_e \in R$ gets added to the tree, then an error location is reached, and $\mathsf{RefineTREE}(n_e)$ is invoked. The abstraction refinement procedure terminates when either a fixpoint is reached or refinement finds a true counterexample.

An important property of procedure $\mathsf{AbstractTREE}$, that stems from its depth-wise exploration of nodes is: *if the loop at line* 3 *gets executed with $i = k$, the program being analyzed has no true counterexamples of length $\leq k$.* This can be proved by induction on k (refer appendix).

Procedure $\mathsf{RefineTREE}$ takes an error node n_e as input and analyzes the counterexample represented by the path from n_0 to n_e in the abstract computation tree. It either confirms the counterexample as true or finds a node for refinement. It initializes the error state at n_e to $\top$ and then uses the abstract $\mathsf{Preimage}$ operation to propagate this error state backward. At a node $n' = \langle l', s' \rangle$ with parent $n = \langle l, s \rangle$, if ψ denotes the backward propagated error state at n', and if $\psi \sqcap s' \neq \bot$, the procedure proceeds in one of the following ways:
(1) Suppose the exact image of the parent, i.e. $\mathsf{Image}(s, l, l')$, does not intersect ψ. Then l' must be a loophead and s' must have been computed as $s'' \nabla_T (s'' \sqcup \mathsf{Image}(s, l, l'))$, where s'' is $\mathsf{Sel}(S_{l',n})$. Furthermore, abstract state s'' cannot intersect ψ, as otherwise, a true counterexample shorter than the current one can be found – an impossibility by the above mentioned property of $\mathsf{AbstractTREE}$. If $s'' \sqcup \mathsf{Image}(s, l, l')$ doesn't intersect ψ, neither s'' nor $\mathsf{Image}(s, l, l')$ intersects ψ. We refine ∇_T by computing an interpolant between $s'' \sqcup \mathsf{Image}(s, l, l')$ and ψ, and including it in T, to make ∇_T precise enough. The abstract state s' is refined by using the refined ∇_T operation. Lemma 1 ensures that this refined state does not intersect ψ. If $s'' \sqcup \mathsf{Image}(s, l, l')$ intersects ψ we simply refine the abstract state to $\mathsf{Image}(s, l, l')$. This is a valid refinement as the image does not intersect ψ (checked in line 4 of $\mathsf{RefineTREE}$). Note also that this refinement implicitly converts $s'' \nabla_T (s'' \sqcup img)$ to a disjunction of s'' and img, with the disjuncts stored at distinct nodes (with location l') in the tree. This differs from [11] where a set of base abstract domain elements is stored

at each node to represent their disjunction. Note that our way of representing disjunctions may result in a node n with location l' having multiple maximal nodes among its predecessors with location l'. After refining node n', we delete its descendents since the abstract states at these nodes may change because of this refinement.

(2) If the exact image $\mathsf{Image}(s, l, l')$ intersects ψ, the abstract error ψ at node n' is propagated backward by the Preimage operation until either a refinement node is identified or n_0 is reached. Since Image and Preimage are exact, $\mathsf{Image}(s, l, l')$ intersects ψ if and only if s intersects $\mathsf{Preimage}(\psi, l, l')$. Therefore it is not necessary to intersect $\mathsf{Image}(s, l, l')$ with ψ before propagating the error backwards. If n_0 is reached during backward propagation we report a true counterexample.

Note that if the Preimage operation is overapproximating, a counterexample reported by RefineTREE may not be a true counterexample. However, if a program is reported to be correct, it is indeed correct. If the Image operation is overapproximating, then a spurious counterexample may not be eliminated because RefineTREE only refines join and widen operations. Consequently AbstractTREE may loop indefinitely, a problem typical of all counterexample guided abstraction refinement tools. This can be rectified by letting RefineTREE improve the precision of Image as well.

3 DAG Refinement

The tree based abstraction refinement technique discussed in the previous section potentially suffers from explosion of paths during forward exploration. Yet another drawback of tree based exploration is that every invocation of RefineTREE analyzes a single counterexample. We propose to address both these drawbacks by adapting the tree based technique of the previous section to work with a DAG. In such a scheme, the abstract computation joins states computed along different paths to the same merge location. It then represents the merged state at a single node in the DAG, instead of creating separate nodes as in a tree based scheme. Subsequently, if it is discovered that merging led to an imprecision that generated a spurious counterexample, the refinement procedure splits the merged node, so that abstract states computed along different paths are represented separately.

The use of a DAG G to represent the abstract computation implies that when an error location is reached at a node n_e, there are potentially multiple paths from the root n_0 to n_e in G. Let the subgraph of G containing all paths from n_0 to n_e be called the counterexample-DAG G_e. Unlike in a tree based procedure, we must now analyze *all* paths in G_e to determine if n_e is reachable. The refinement procedure either finds a true counterexample in G_e, or if all counterexamples are spurious, it replaces a set of imprecise operations by more precise ones along every path in G_e. Refinement proceeds by first computing a *set of abstract error preimages*, $err(n)$, at each node n in G_e. For a node n, $err(n)$ is computed as the set union of the preimage of every element in $err(n')$, for every successor n' of n in G_e.

Unlike in a tree based procedure, a node $n' = \langle l', s' \rangle$ may not have a unique predecessor in the counterexample DAG G_e. We say that node n' is a *refinement* *node with respect to predecessor* $n = \langle l, s\rangle$ if $\exists e' \in err(n'), s' \sqcap e' \neq \bot$ and $\forall e \in err(n), s \sqcap e = \bot$. The goal of refinement at such a node n' is to improve the precision of computation of s' from s, so that the new abstract state at n' does not intersect $err(n')$. However, the abstract states already computed at descendents of n' in G may be rendered inexact by this refinement, and may continue to intersect the corresponding abstract error preimages. Hence we delete all descendents of n' in G.

Refinement is done at node $n' = \langle l', s'\rangle$ in one of the following ways: (i) If l' is a merge location and n' has predecessors $n_1 = \langle l_1, s_1\rangle, \ldots, n_k = \langle l_k, s_k\rangle$, then refinement first deletes deletes n' and all its incoming edges. Then it creates k new nodes $m_1, \ldots, m_k$, where $m_i = \langle l', t_i\rangle$ and $t_i = \mathsf{Image}(s_i, l_i, l')$. (ii) If l' is a loophead, then as done in Algorithm RefineTREE, refinement either introduces disjunctions (implicitly) or does interpolated widen with a refined set of interpolants. An interpolant is computed between the joined result at n' and each of the abstract error states from the set $err(n')$. The result of the interpolated widen is guaranteed not to intersect $err(n')$.

Consider a merge node n' that is a refinement node with respect to predecessor n but not with respect to predecessor m. Suppose no ancestor of n is a refinement node while m has an ancestor p that is a refinement node. In this case if we apply refinement at p before n', then node n' will be deleted and no counterexample corresponding to a path through n and n' would have any of its nodes refined. To prevent this, nodes are refined in reverse topological order. This ensures that at least one node along each path in the counterexample-DAG is refined.

Lemma 3. *Let $n_e = \langle l_e, s_e\rangle$ be a node in a counterexample-*DAG *G_e corresponding to error location l_e. Every invocation of refinement with G_e either finds a true counterexample or reduces the number of imprecise operations on each spurious counterexample ending at n_e in G_e.*

As discussed above, the abstraction procedure aggressively merges paths at merge locations, and refinement procedure aggressively splits paths at merge locations. One could, however, implement alternative strategies, where we selectively merge paths during abstraction and selectively split them during refinement. For example, whenever refinement splits a merge node n into nodes $n_1, \ldots n_k$, it may be useful to remember that descendents of n_i should not be merged with those of n_j where $i \neq j$ during forward exploration. This information can then be used during abstraction to selectively merge paths leading to the same merge location. Our implementation uses this simple heuristic to prevent aggressive merging of paths. Note also that as an optimization, we store and propagate back only those abstract error preimages s'_e at node $n' = \langle l', s'\rangle$ that satisfy $s'_e \sqcap s' \neq \bot$. This potentially helps in avoiding an exponential blow up of error preimages during refinement.

Progress Guarantees: It would be desirable to prove that our DAG-based abstraction refinement scheme has the following *progress* property: *Once a counterexample is eliminated it remains eliminated forever.* There are two reasons why the abstraction refinement procedure may not ensure this. Firstly, it does not keep track of refinements performed earlier, and secondly, the interpolated widen operation is in general non monotone, i.e., $A' \sqsubseteq A$ and $B' \sqsubseteq B$ does not necessarily imply $(A' \nabla_T B') \sqsubseteq (A \nabla_T B)$. Progress can be ensured by keeping track of all earlier refinements and by using monotone operations. We propose addressing both these issues by using a $Hint$ DAG H, which is a generalization of the list based *hints* used in [11]. Monotonicity of interpolated widen is ensured by intersection with the corresponding widened result in the previous abstraction iteration. The details of using $Hint$ DAG can be found in appendix. Lemma 3 along with the use of Hint DAG ensures the following: *a counterexample c having k imprecise operations is eliminated in at most k refinement iterations with counterexample-DAGs containing c*. The $Hint$ DAG also ensures that *once a counterexample is eliminated, it remains eliminated in all subsequent iterations of abstraction*.

4 Implementation

We have implemented our algorithm in a tool, Dagger, for proving assertions in C programs. Dagger is written in ocaml and uses the CIL [6] infrastructure for parsing our input programs. Dagger uses the octagon and polyhedra abstract domains as implemented in the Apron library [1]. We use flow insensitive pointer analysis provided by CIL to resolve pointer aliases in a sound manner.

We have implemented several algorithms for abstract computation which include the TREE and DAG based exploration with and without interpolated widen. This is done to compare the enhancements provided by each of these techniques. Dagger keeps track of a separate interpolant set for each program location as opposed to having a single monolithic set of interpolants for all program locations. We outline the interpolation algorithms for the octagon and polyhedra abstract domains, and then explain an optimization of caching abstract error preimages.

Interpolation for the octagon domain. In the octagon abstract domain every non-$\bot$ abstract element is represented by a set of constraints of the form $l \bowtie e \bowtie u$, where $\bowtie \in \{<, \leq\}$, l and u are real or rational constants and e is an expression that is either a single variable, difference of two variables or sum of two variables. We will assume that the set of constraints is in canonical form, i.e., l and u are tight bounds for the expression e. InterpolateOct computes an interpolant I of two non-$\bot$ canonical octagons A and B such that $A \sqcap B = \bot$. This takes time quadratic in the number of program variables. Note that in Algorithm ApplyRefinement (Figure 2) canonicalization would already have been done at line 3 when checking the emptiness of intersection, before InterpolateOct is invoked at line 4.

Interpolation for the polyhedra domain. In this domain, each non-$\bot$ abstract element is represented by a set of non redundant constraints. For an abstract element A, let $var(A)$ be the set of variables occurring in the constraints of A. Function $Project(A, V)$, computes the projection of polyhedra A on a set of variables V, i.e., it existentially quantifies the variables not in V. Given two non-$\bot$ polyhedra A and B such that $A \sqcap B = \bot$, the interpolant can be computed as below.

InterpolateOct **(A,B)**

$I \leftarrow \emptyset$
for all expressions e **do**
  Let $al : l_a \bowtie e$ and $au : e \bowtie u_a$ be constraints in A
  Let $bl : l_b \bowtie e$ and $bu : e \bowtie u_b$ be constraints in B
  if $au \sqsubseteq \{\neg bl\}$ **then**
    $I \leftarrow I \cup \{\neg bl\}$
  if $al \sqsubseteq \{\neg bu\}$ **then**
    $I \leftarrow I \cup \{\neg bu\}$

InterpolatePoly1 **(A,B)**

$I \leftarrow \emptyset$
for all constraints c in B **do**
  if $A \sqsubseteq \{\neg c\}$
    $I \leftarrow I \cup \{\neg c\}$

InterpolatePoly2 **(A,B)**

$V \leftarrow var(A) \cap var(B)$
$I \leftarrow Project(A, V)$

InterpolatePoly1 computes an interpolant from the constraints of B. Any $i \in I$ computed by InterpolatePoly1 is implied by A and does not intersect B. It has variables common to the constraints of A and B. Note that there may not be any constraint c in B whose negation is implied by A. In such a case, we obtain interpolants by algorithm InterpolatePoly2. In our implementation, we first try to get an interpolant by InterpolatePoly1 algorithm. If no interpolant is found, then we use InterpolatePoly2. As part of future work, we wish to incorporate interpolation techniques from [16,19] in our tool DAGGER. The correctness proofs and complexity analysis of InterpolatePoly1 and InterpolatePoly2 algorithms are given in appendix.

In each of the above mentioned abstract domains, a non-$\perp$ abstract element is represented as a set of constraints (conjoined implicitly). For any two abstract elements A and B, a simple interpolated widen operator can be defined as: $A \nabla_T B = B$ if $A = \perp$. Otherwise $A \nabla_T B = \{c \in T \mid \gamma(A) \subseteq \gamma(\{c\}) \wedge \gamma(B) \subseteq \gamma(\{c\})\} \cup \{c \in A \mid \gamma(B) \subseteq \gamma(\{c\})\}$.

Caching error states. Our implementation also uses an additional optimization of caching abstract error preimages at refinement points. This optimization has been empirically found to be useful in early detection of imprecisions that lead to errors in future explorations. Compared to [11] where widen is refined by join, the use of interpolated widen can potentially increase the total number of image and preimage computations in the overall abstraction refinement loop. Caching abstract error preimages helps in mitigating this effect (see appendix for further discussion of this optimization technique).

Experimental Evaluation. We have evaluated our implementation on the suite of buffer overflow programs (adapted from Sendmail) developed by Zitser et. al. [22], the set of STING benchmarks [20], and a miscellaneous set of programs [1].

Our current implementation is intra procedural and we handle multiple procedures by providing procedure summaries by way of annotations. The experiments are performed on an Intel(R) Xeon 3.00 GHz processor with 4GB RAM. The experimental results are given in Table 1.

Benchmark programs. The Sendmail programs have nested while loops with branching structures within the loops. The assertions in these programs check bounds on array accesses. Programs with 'ok' suffix are correct and those with 'bad' suffix have array bound errors. The programs p1-bad and p3-bad have deep counterexamples whose lengths depend on the size of the array. The STING programs have a single while loop with nondeterministic branching in the loop

[1] all programs available at http://www.cfdvs.iitb.ac.in/~bhargav/dagger.html

	Dagger		TREE + ∇		TREE + ∇_I		DAG + ∇		DAG + ∇_I		Blast		Slam		GR06		Armc	
Pgm	I	II	I	II	I	II	I	II	I	II	I	II	I	II	I	II	I	II
Sendmail																		
p1-ok	4.64	9	1940	18408	11.2	16	131.5	412	5.76	9	*	*	*	*	*	*	-	-
p2-ok	0.27	4	35.8	3996	0.77	4	64.23	1332	0.39	4	*	*	*	*	*	*	*	*
p3-ok	0.15	0	18.4	1	18.3	1	0.15	0	0.14	0	2.2	4	*	*	*	*	-	-
p1-bad	3.31	11	2.81	33	8	17	3.53	19	21.7	38	1368	46	*	*	12	33	-	-
p2-bad	0.06	1	0.08	1	0.08	1	0.06	1	0.06	1	1.1	7	9.9	12	0.1	1	0.12	1
p3-bad	4.91	49	1735	2402	252	203	30.85	64	101	53	*	*	*	*	*	*	-	-
StInG																		
seesaw	0.04	2	*	*	0.05	2	*	*	0.05	2	!	!	1.0	1	0.82	6	*	*
bkley	0.04	1	0.04	0	0.04	0	0.05	1	0.04	1	!	!	2.90	5	0.10	2	4	16
bk-nat	0.06	2	0.09	2	0.06	1	0.11	4	0.07	2	!	!	!	!	0.43	3	3.25	18
hsort	0.14	3	*	*	0.16	3	*	*	0.14	3	!	!	1.10	1	0.72	3	22.5	40
efm	0.09	1	0.09	1	0.09	1	0.08	1	0.09	1	!	!	1.40	1	0.06	0	16.9	35
lifo	0.31	2	0.57	2	0.55	2	0.38	3	0.32	2	!	!	7.50	9	3.3	6	75.3	88
lifnat	0.49	3	*	*	1.46	5	*	*	0.48	3	!	!	6.60	9	29.55	12	!	!
cars	19.5	8	*	*	17.6	8	*	*	19.5	8	!	!	1.80	3	*	*	107	27
barbr	3.46	8	*	*	4.80	7	*	*	3.44	8	!	!	43.9	22	10.5	6	674	205
swim	0.60	2	2.46	13	0.60	2	1.49	9	0.60	2	!	!	!	!	11.1	6	579	137
swim1	0.72	3	2.57	13	0.79	3	1.54	9	0.72	3	!	!	!	!	11.2	6	767	144
hsort1	0.07	1	*	*	0.08	1	*	*	0.15	1	!	!	1.3	1	*	*	0.15	1
barbr1	0.63	2	*	*	1.15	2	*	*	0.62	2	!	!	16.1	11	*	*	570	109
lifnat1	0.59	6	*	*	8.71	23	*	*	0.74	5	!	!	!	!	*	*	!	!
Miscellaneous																		
f1a	0.01	0	0.01	0	0.01	0	0.01	0	0.01	0	1.07	12	*	*	0.01	0	*	*
ex1	0.04	1	*	*	0.06	1	*	*	0.04	1	!	!	!	!	*	*	0.62	3
f2	0.07	1	*	*	0.06	1	*	*	0.07	1	!	!	!	!	0.42	2	*	*
ex2	0.03	0	5.4	0	5.4	0	0.03	0	0.03	0	506	132	*	*	5.4	0	1.7	12
JM06	0.02	0	0.02	0	0.02	0	0.02	0	0.02	0	*	*	*	*	0.02	0	*	*
Programs Strengthened with loop invariants obtained by initial widen pass																		
p1-ok'	4.64	9	1948	18413	8.2	13	121	411	4.76	9	200	24	*	*	*	*	-	-
p2-ok'	0.27	4	35.8	3996	0.77	4	64.23	1332	0.39	4	*	*	*	*	*	*	*	*
JM06'	0.02	0	0.02	0	0.02	0	0.02	0	0.02	0	0.07	2	*	*	0.02	0	0.17	1
barbr'	3.46	8	*	*	4.80	7	*	*	3.44	8	!	!	4.7	14	10.5	6	890	153
barbr1'	0.63	2	*	*	1.15	2	*	*	0.62	2	!	!	1.7	5	*	*	767	104
lifnat1'	0.59	6	*	*	6.71	10	*	*	0.74	5	!	!	!	!	*	*	205	71

Table 1. Experimental results. Column I: time (seconds), Column II: number of refinement iterations. '*' denotes non-termination in 2000 sec, '!' denotes inability of tool to discover new predicates, and '-' denotes tool crash.

body. We modified the examples to assert for the invariants computed by StInG. For the programs hsort1, barbr1 and lifnat1, we dropped some conjuncts in the invariants computed by StInG while writing the assertion. Program ex2 has a sequence of if-then-else statements, leading to an exponential explosion of paths. Program JM06 is the benchmark program in [16] that could not be analyzed by Blast. The programs with primed names in the last six rows were obtained by annotating the corresponding unprimed programs with location invariants obtained from an initial widen based analysis (using "assume" statements) as suggested in [15] suggests. All Sendmail programs and JM06 were analyzed in Dagger using the octagon abstract domain. All StInG programs and f1a, f2, ex1 and ex2 were analyzed in Dagger using the polyhedra abstract domain.

Description of columns. In Table 1 we compare Dagger with other abstraction refinement tools (Slam, Blast, Armc), with our earlier tool GR06 [11], and with combinations of Dagger's constituent optimizations. We could not compare with Impact and with the tools mentioned in [15, 21] due to their unavailability. The column DAGGER gives results for a DAG based exploration, as described in Section 3, with the additional optimization of caching abstract error preimages at refinement points. The column TREE + ∇ is for a TREE based exploration with widen refined by $\sqcup$ instead of by interpolated widen. The column TREE + ∇_I gives results for a TREE based exploration, as discussed in Section 2. Similarly, the column DAG + ∇ gives results for a DAG based exploration with widen refined by $\sqcup$ instead of by interpolated widen. The column DAG + ∇_I gives results for a DAG based exploration, as discussed in Section 3.

Advantages of interpolated widen. To understand the effect of interpolated widen, we compare the columns TREE + ∇ and DAG + ∇, where interpolated widen is not used, with the corresponding columns TREE + ∇_I and DAG + ∇_I, where interpolated widen is used. The programs seesaw, hsort, lifnat, cars, barbr, hsort1, barbr1, lifnat1, ex1 and f2 require interpolated widen to compute inductive invariants strong enough to prove the desired properties. For p1-ok and p2-ok, exploration without interpolated widen performs a large number of refinement iterations proportional to the size of the array being processed, as seen in columns TREE + ∇ and DAG + ∇. Interpolated widen eliminates the dependence on array size, as seen in columns TREE + ∇_I and DAG + ∇_I.

Advantages of DAG exploration. For programs p3-ok and ex2, TREE based exploration explores exponentially many paths. However, DAG based exploration avoids this blow up by merging abstract states along different paths at each merge location. DAG based exploration is also effective in detecting true counterexamples. In p3-bad, the TREE based technique explores several spurious counterexamples before discovering a true error. The DAG based technique reduces this effort significantly. It also does not blow up while analyzing multiple counterexamples by backward propagation of error. Interestingly, in p1-bad, the TREE based exploration got lucky and found a true counterexample quickly. The STING benchmarks do not have significant branching structure. Thus, the DAG + ∇ and TREE + ∇ techniques, and also DAG + ∇_I and TREE + ∇_I explorations perform similarly for these examples.

Advantages of caching. For programs p1-bad and p3-bad, the number of image and preimage computations using TREE + ∇_I and DAG + ∇_I grows quadratically with the length of the counterexample. This contrasts with TREE + ∇ and DAG + ∇, where this number grows linearly with the length of the counterexample, leading to much lesser computation times. This discrepancy arises because in TREE + ∇ and DAG + ∇, widens are refined to joins that are more precise than interpolated widens. Thus, once a widen is refined, it does not need to be refined further. By caching error preimages at refinement points, the above drawback can be significantly addressed in interpolated widen based techniques. This can be seen by comparing the DAGGER column with the DAG + ∇_I column for programs p1-bad and p3-bad.

Comparison with other refinement tools. For the Sendmail examples p1-ok, p2-ok, p3-bad and for JM06, none of SLAM, BLAST, GR06 and ARMC are able to find the right predicates. DAGGER's interpolated widen however finds the right predicates in a few iterations. On most of Sendmail examples, GR06 does not terminate due to an explosion in the number of disjuncts. When location invariants obtained from an initial widen based analysis are added to the original program, the performance of other tools does not always improve. The last six rows of Table 1 illustrate this. For BLAST and SLAM, the performance improves for some programs by way of either terminating within 2000s (where it did not terminate earlier), or faster convergence. However for other programs (p2-ok' for BLAST, and p1-ok', p2-ok', JM06' for SLAM) these tools still do not terminate in 2000s. For ARMC the performance improves on some examples (lifnat', JM06') and degrades on others (barbr', barbr1'). For GR06 and DAGGER the performance does not significantly change after adding invariants since these tools can easily discover these invariants. This illustrates that invariants obtained from an initial widen based analysis may be too weak to help refinement, and that interaction between widen and interpolation as implemented in DAGGER is useful.

The refinement engine of BLAST fails for StInG programs as it is equipped to generate only difference and bounds constraints, while the StInG programs need more expressive invariants. SLAM is unable to make progress on bk-nat, swim, swim1 and lifnat1, as it cannot discover the correct predicates. ARMC takes several more iterations (and longer execution times) compared to DAGGER to generate the right predicates on most of the StInG examples. However for the programs seesaw, lifnat and lifnat1, it is unable to generate the right predicates, and hence does not terminate. GR06 is able to compute the correct inductive invariants for many programs in the StInG benchmarks. But the programs cars, hsort1, barbr1, and lifnat1 fail with this technique. DAGGER and the constituents of DAGGER that use interpolated widen (namely $\mathsf{TREE} + \nabla_I$ and $\mathsf{DAG} + \nabla_I$) are able to prove these programs correct in a small number of iterations.

Finally, looking at the miscellaneous benchmarks, we find that SLAM fails on all these examples. BLAST fails on ex1, f2 and JM06. GR06 fails on ex1, and ARMC fails on f1a, f2 and JM06. Again, DAGGER and the constituents of DAGGER that use interpolated widen (namely $\mathsf{TREE} + \nabla_I$ and $\mathsf{DAG} + \nabla_I$) are able to prove these programs correct in a small number of iterations.

Tools like BLAST, SLAM, and ARMC use techniques beyond what we have discussed for widen and interpolants. For example, BLAST uses recursive enumeration of predicates, SLAM uses several heuristics to determine a good set of predicates, and both BLAST and ARMC use several sophisticated algorithms to compute interpolants. In contrast, DAGGER uses very simple widen and interpolation operators, and by combining these appropriately (and dynamically), it outperforms these other tools.

5 Conclusion

We presented three new techniques to automatically refine abstract interpretations to tune the precision of fixpoint computations dynamically and reduce the number of false errors produced by abstract interpretation. We have proved that our refinements guarantee progress in a formal sense. However, since assertion checking is undecidable, our procedure is not guaranteed to terminate. In practice, we find that our procedure terminates and outperforms tools available to us on a variety of benchmarks. Though our implementation DAGGER uses polyhedra and octagons, our techniques can be used with any choice of abstract domain, widen, join and interpolation operators.

Acknowledgments. The first author was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.

References

1. APRON. Numerical Abstract Domain Library. http://apron.cri.ensmp.fr/library/.
2. R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. Technical Report 344, University of Parma, Italy, 2004.
3. R. Bagnara, P.M. Hill, E. Ricci, and E. Zaffanella. Precise widening opertors for convex polyhedra. In *SAS 03*, 2003.
4. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02*, pages 1–3, 2002.
5. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.
6. CIL. Infrastructure for C Program Analysis and Transformation. http://manju.cs.berkeley.edu/cil/.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL 77*, pages 238–252, 1977.
8. J. Esparza, S. Kiefer, and S. Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In *TACAS 06*, pages 489–503, 2006.
9. J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *ESEC/SIGSOFT FSE*, pages 227–236, 2005.
10. D. Gopan and T. W. Reps. Lookahead widening. In *CAV 06*, pages 452–466, 2006.
11. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS 06*, pages 474–488, 2006.
12. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *FMSD*, 11(2):157–185, August 1997.
13. T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, 2004.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
15. H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV*, 2006.
16. R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.
17. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
18. A. Rybalchenko and A. Podelski. Armc: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
19. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.
20. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear-relations analysis. In *SAS*. Springer-Verlag, 2004.
21. C. Wang, Z. Yang, A. Gupta, and F. Ivancic. Using counterexamples for improving the precision of reachability computation with polyhedra. In *CAV*, 2007.
22. M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *FSE*, pages 97–106, 2004.

A Appendix

A.1 Proofs

Proof of Lemma 2: Since ∇_T is a widen operator on the base abstract domain $\Sigma^\sharp$, the reflexive, transitive closure of the relation $\{(d_1, d) \in \Sigma^\sharp \times \Sigma^\sharp \mid \exists d_2 \in \Sigma^\sharp, d_1 \sqsubset d_2 \wedge d = d_1 \nabla_T d_2\}$ defines a partial ordering $\vdash_{\nabla_T}$ on $\Sigma^\sharp$ that satisfies the ascending chain condition [2]. Let $S_0 = \{a_{0,1}, \ldots, a_{0,r}\}$. From the definition of ∇_T^p, it is straightforward to show by induction on i that (i) $|S_i| \leq |S_0|$ for all $i \geq 0$, and (ii) for every $a_{i+1,n} \in S_{i+1}$, where $i \geq 0$, there exists $a_{i,m} \in S_i$ such that $a_{i,m} \vdash_{\nabla_T} a_{i+1,n}$. Since $\vdash_{\nabla_T}$ is transitive, it follows that for every $a_{i,m} \in S_i$, there exists $a_{0,p} \in S_0$ such that $a_{0,p} \vdash_{\nabla_T} a_{i,m}$. Thus, every element of every S_i ($i \geq 0$) must be in a non-decreasing chain starting from some element in S_0. Combining the facts that (i) there are a finite number of elements in S_0, (ii) $\vdash_{\nabla_T}$ satisfies the ascending chain condition, and (iii) $|S_i| \leq |S_0|$, it follows immediately that there exists $u \geq 0$ such that $S_v = S_u$ for all $v \geq u$.
Property of AbstractTREE: If the loop at line 3 gets executed with $i = k$, the program being analyzed has no true counterexamples of length $\leq k$.
Proof: Proof by induction on k. Since $l_0 \notin R$, the property holds initially when $k = 0$. Suppose the loop executes with $i = k$, where $k \geq 0$, and there are no true counterexamples of length $\leq k$. After the loop terminates, there are two possibilities: (i) if no error node is found at line 11, there is no true counterexample of length $k+1$. This, in conjunction with the induction hypothesis, implies that when i is incremented in line 16, and the loop at line 3 resumed with $i = k+1$, the property holds. (ii) if an error node at depth $k+1$ is found at line 11, RefineTREE is called. Since RefineTREE always returns an integer at least one less than the depth of the error node, when the loop at line 3 of AbstractTREE is resumed, i is no larger than k. By the induction hypothesis, the property holds again.

A.2 Progress Guarantees

A *Hint* DAG H, is a generalization of the list based *hints* used in [11]. Like in DAG G, every node in H has a location and abstract state associated with it. As G is explored and refined, we maintain the invariant that G is a subgraph of H. Thus, there is a corrspondence map *corr* from nodes in G to nodes in H. For a node $n = \langle l, s\rangle$ in G such that $corr(n) = h = \langle l', s'\rangle$ in H we have that $l = l'$ and $s \sqsubseteq s'$.

Initially, H has just one node $h_0 = \langle l_0, \top\rangle$. The node n_0 in G corresponds to h_0. The abstraction procedure refers to H when adding new nodes and edges in G. If the abstract computation for a new node n is already represented in H (possibly because of an earlier exploration or refinement), it mimics that computation and maps the new node created in G to the corresponding node h in H. The abstract state computed at n in G is then intersected ($\sqcap$ operation) with the abstract state at h. The result is stored as the abstract state at h and also at n. If, however, the abstract computation for a new node n is not already represented in H, abstaction computation proceeds as was done earlier (without the hint dag), and then it creates corresponding nodes and edges in H to record this abstract computation. When refinement is done at a node n in G, refinement procedure records this refinement in H. This includes updating the structure of H to record conversion of joins to splits. In addition every node h in H corresponding to a refined node n in G stores the refined abstract state of n. While nodes in G may be deleted by DeleteDescendents during refinement, the corresponding nodes in H are never deleted. Thus the DAG H monotonically grows.

Theorem 1. *The Hint* DAG *ensures that once a counterexample is eliminated, it remains eliminated in all subsequent abstraction iterations.*

Proof. The $Hint$ DAG H as described above ensures the following: (a) Every time a counterexample is eliminated in G, it is recorded in H by virtue of the construction of H, and (b) Every counterexample eliminated in H can never be encountered in subsequent iterations of abstraction and refinement, due to the intersection of abstract states in G with those stored in corresponding nodes in H.

Since all operations other than interpolated widen are monotone, a $Hint$ DAG that stores abstract states only at loophead nodes also gives the same guarantee as in Theorem 1. Given a node n_e with an error location in DAG G, let G_e be the corresponding counterexample-DAG. Every path in G_e corresponds to a potential counterexample. From Lemma 3 and the $Hint$ DAG as described above, it follows that *a spurious counterexample c in G_e having k imprecise operations is eliminated in at most k refinements that analyze the counterexample-*DAG *containing the counterexample c.*

A.3 Caching of abstract error preimages

Caching the abstract error states obtained during preimage computation at the refinement points can help early detection of imprecisions that lead to the error in future. For example, consider the incorrect program: `x = 0; while (x < 100) x = x+1; assert (x > 100).` The interpolated widen algorithms as described above need to perform of the order of 100^2 Preimage computations. Caching the computed Preimage at refinement points enables us to find the error with a linear number of Preimage computations. We find that this optimization is greatly helpful when used with interpolated widen.

A.4 Details of interpolation algorithms

Proof that InterpolateOct computes an interpolant: Every constraint c that is collected in I refers to variables that appear in a constraint of A and a constraint of B. Further, every such constraint c satisfies $\gamma(A) \subseteq \gamma(\{c\})$ and $\gamma(\{c\}) \cap \gamma(B) = \emptyset$. Therefore, $\gamma(A) \subseteq \gamma(I)$ and $\gamma(I) \cap \gamma(B) = \emptyset$. As A and B

are in canonical form, this implies that $A \sqsubseteq I$ and $I \sqcap B = \bot$. Hence I is an interpolant of A and B.

Proof that InterpolatePoly2 computes an interpolant I: I has constraints over common variables of A and B. Let the constraints of A be over the set of variables $V \cup V_a$, and those of B be over set of variables $V \cup V_b$ such that $V_a \cap V_b = \emptyset$. Let the set of variables of program be $V \cup V_a \cup V_b$. Any abstract element over these variables represents a set of points $p = (p_v, p_a, p_b)$, where p_v, p_a, and p_b is the valuation to variables in the set V, V_a, and V_b respectively. Let $A = \{(p_v, p_a, p_b) \mid (p_v, p_a)$ satisfies constraints of $A\}$. Then $A \downarrow V = \{(p_v, p_a, p_b) \mid \exists p'_a, (p_v, p'_a)$ satisfies constraints of $A\}$. Let (p_v, p_a, p_b) be a point common to $A \downarrow V$ and B. Then there exists another point $p = (p_v, p'_a, p_b)$ such that (p_v, p'_a) satisfies constraints of A (by the definition of projection operation). Since (p_v, p_b) satisfies constraints of B, $p \in B$. Thus p belongs to both A and B – a contradiction. Thus there cannot be a point common to $A \downarrow V (= I)$ and B. Hence $I \sqcap B = \bot$ and $A \sqsubseteq I$, by definition of projection operation. Hence I is an interpolant of A and B.

Complexity of InterpolatePoly1 and InterpolatePoly2 Polyhedra are typically represented by both the constraint form, and the parametric form (points, rays and lines). This is known as dual representation. Let n_a and m_a be the size of the constraints representation and parametric representation respectively of polyhedra A. Similarly, n_b and m_b be the size of the constraints and parametric representation respectively of polyhedra B.

For each constraint c of B the implication check at line 4 of InterpolatePoly1 can be done in time $O(m_a)$. Hence the time complexity of the algorithm InterpolatePoly1 is $O(n_b m_a)$. A less efficient but more effective version of InterpolatePoly1 first performs projection of B on the variables common to both A and B. This version, InterpolatePoly1', is given below.

InterpolatePoly1' **(A,B)**
$I \leftarrow \emptyset$
$B \leftarrow Project(B, var(A) \cap var(B))$
for all constraints c in B **do**
  if $A \sqsubseteq \{\neg c\}$
    $I \leftarrow I \cup \{\neg c\}$

InterpolatePoly1' may be able to find an interpolant when InterpolatePoly1 fails to find one. In this sense it is more effective than InterpolatePoly1, but its worst case complexity is same as that of InterpolatePoly2.

Let d be the dimension of the polyhedra which typically corresponds to the number of variables in the program. The complexity of the projection operation and hence that of InterpolatePoly2 is $O(n_a^{d/2})$ where n_a is the number of constraints of the polyhedra A and d is its dimension.