Counterexample Driven Refinement for Abstract Interpretation

Bhargav S. Gulavani[1] and Sriram K. Rajamani[2]

[1] IIT Bombay
[2] 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 a new counterexample driven refinement technique to reduce false errors in abstract interpretations. Our technique keeps track of the precision losses during forward fixpoint computation, and does a precise backward propagation from the error to either confirm the error as a true error, or identify a refinement so as to avoid the false error.

Our technique is quite simple, and is independent of the specific abstract domain used. An implementation of our technique for affine transition systems is able to prove invariants generated by the StInG tool [19] without doing any specialized analysis for linear relations. Thus, we hope that the technique can work for other abstract domains as well. We sketch how our technique can be used to perform shape analysis by simply defining an appropriate widening operator over shape graphs.

1 Introduction

Abstract interpretation [8] is a generic technique to compute sound fixpoints for programs. Suppose we are interested in checking if a program satisfies invariant φ. If the fixpoint computed by an abstract interpretation of the program P satisfies φ, then we know that all concrete behaviors of the program satisfy φ. However, if such a fixpoint does not satisfy the property φ, then there are two possibilities: (1) the program does not satisfy φ (we have found a "true error" in the program), or (2) the program indeed satisfies the property φ, but the abstract interpretation was not precise enough to verify it (we have found a "false error" in the program). Losing precision while computing fixpoints is inevitable if we want to analyze programs with infinite domains, or scale the analysis to large programs. However, losing too much precision leads to too many false errors and reduces usability of the analysis tool.

Predicate abstraction [10] is a particular form of abstract interpretation. Tools based on predicate abstraction to verify finite state interface protocols on programs have become popular over the past few years [4, 12, 6]. In order to reduce false errors, these tools analyze an abstract counterexample to check if the counterexample is feasible in the concrete program. If the counterexample is infeasible they add more predicates to improve precision of predicate abstraction. This process, called *counterexample driven refinement* continues iteratively until (1) the

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 474–488, 2006.

property is proved, or (2) a true error is found, or (3) either time or memory is exhausted [14, 7].

Abstract interpretations operate over lattices, and compute overapproximations to semantics of programs as fixpoints. Such fixpoint computations may not converge if the lattice has infinite ascending chains. Widening is a technique used to ensure convergence of fixpoint computations. The widening operator ∇ has the property that for all x and y the result $x \nabla y$ is greater than both x and y. Furthermore, widening guarantees convergence of fixpoint computation in the following sense. Given any infinite increasing sequence $x_0, x_1, x_2, \ldots$, the sequence $y_0, y_1, y_2, \ldots$ given by $y_0 = x_0$ and $y_{i+1} = y_i \nabla (y_i \cup x_{i+1})$ is guaranteed to converge. Examples of widening operators on polyhedral domains can be found in [9, 3].

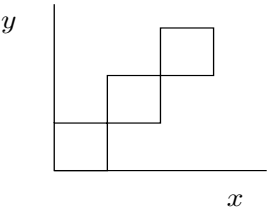
In this paper, we present a new counterexample driven refinement that can be used to reduce false errors in any abstract interpretation. Precision loss in abstract interpretation occurs primarily due to widen operators. We parameterize the abstract interpreter with a set of hints, which specify the steps in the fixpoint computation where more precise operators should be used in place of widen. Initially the set of hints is empty. We analyze spurious counterexamples and make additions to the set of hints, thereby guiding the fixpoint to be as precise as necessary to prove the property of interest. Furthermore, powerset domains can add further precision to abstract interpretation. However, powerset domains do not scale to large programs without aggressive use of widening. We describe how counterexample driven refinement can be applied to powerset domains. The key idea here is a new connector that allows lifting a widening operator from a base domain to the corresponding widen operator in a powerset domain. We explain our technique informally using two examples below. A formal description is given in Section 2.

Consider the example program shown in Figure 1(a). For this example, we use the abstract domain of convex polyhedra. First, we perform a symbolic fixpoint computation applying widening every time along the back edge of the while-loop to ensure termination. When the loop head is first encountered, we have the symbolic state $S_0 \stackrel{\Delta}{=} 0 \leq x \leq 2 \wedge 0 \leq y \leq 2$. After executing the loop body once, we get a new set of states $2 \leq x \leq 4 \wedge 2 \leq y \leq 4$. We perform widening to obtain the set $S_1 \stackrel{\Delta}{=} 0 \leq x \wedge 0 \leq y$. It turns out that S_1 is a fixpoint for the loop. However, this loop invariant is not sufficient to ensure that $x \neq 4 \vee y \neq 0$, and the analysis reports that the assertion may fail.

The error state reached by the analysis is $x = 4 \ \wedge \ y = 0$, which is a false error that resulted due to the imprecision in the widening operator. Inspired by approaches to perform counterexample driven refinement for predicate abstraction [7], we propagate this error state backwards, using pre-image computations, and determine that the first application of widening is responsible for the false error, and that using least upper bound (LUB) instead of widening in the first iteration avoids the error. Thus, we add iteration count 1 to the set of hints.

Using the updated hints, we recompute the abstract fixpoint, using the LUB operator (convex hull for convex polyhedral domain) in the first iteration. This

```
assume 0 ≤ x ≤ 2
assume 0 ≤ y ≤ 2
while(*)
  x := x + 2
  y := y + 2
assert !((x = 4) ∧ (y = 0))
```

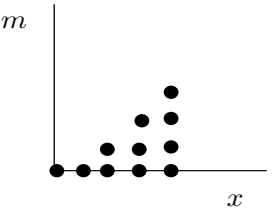


Fix point computed:

$0 \leq x \wedge 0 \leq y \wedge$
$y \leq x + 2 \wedge x \leq y + 2$

(a)

```
x := 0; m := 0
while(x < N)
  if(*)
    m := x
  x := x + 1
if(N > 0)
  assert 0 ≤ m < N
```



Fix point computed:

$0 \leq x \wedge m = 0 \ \vee$
$0 \leq m \wedge m + 1 \leq x$
$\wedge \ x \leq N$

(b)

Fig. 1. (a) Example program that has a stair-case like reachable region (b) Example program that finds the index of the minimum element in an array

results in the set of states $S_1' \stackrel{\Delta}{=} 0 \leq x \leq 4 \wedge 0 \leq y \leq 4 \wedge y \leq x + 2 \wedge x \leq y + 2$. Applying widening after second iteration we get the set of states $S_2' \stackrel{\Delta}{=} 0 \leq x \wedge 0 \leq y \wedge y \leq x + 2 \wedge x \leq y + 2$. This turns out to be the fixpoint for the loop as well and is strong enough to prove the assertion.

Next, consider the example program shown in Figure 1(b). The program searches for the index m of the minimum element in an array of size N. The array contents and the minimum element have been abstracted out, and only the updates to the index variables m and x have been retained. No loop invariant expressed as a single convex polyhedron is strong enough to prove the assertion. Thus, we need to use sets of convex polyhedra as our abstract domain. As we describe below, our technique discovers a disjunctive loop invariant that is strong enough to prove the assertion.

We start by performing symbolic fixpoint computation, applying widening every time along the back edge of the while-loop to ensure termination. When the loop-head is first encountered, we have the set of states $S_0 \stackrel{\Delta}{=} x = 0 \wedge m = 0$. After executing the loop body once, we get a new set of states $x = 1 \ \wedge \ m = 0 \ \wedge \ x \leq N$. We perform widening to obtain the set $S_1 \stackrel{\Delta}{=} x \geq 0 \ \wedge \ m = 0$. The second iteration of the loop produces different states depending on whether the if branch is taken inside the loop: $(x \geq 1 \ \wedge \ x \leq N \ \wedge \ m = 0) \vee (x \leq N \ \wedge \ x = m+1 \wedge m \geq 0)$. Applying widening again, we obtain the set $S_2 \stackrel{\Delta}{=} x \geq 0 \wedge m \geq 0$. It turns out that S_2 is a fixpoint for the loop. However, this loop invariant is not sufficient to ensure that $m < N$, and the analysis reports that the assertion may fail.

The error state reached by the analysis is $m \geq N \ \wedge \ x \geq 0 \ \wedge x \geq N \wedge N > 0$. This is a false error that resulted due to imprecision in the widening operator. Here again we propagate this error state backwards using pre-image computations, and determine that the second application of widening is responsible for the false error, and that using LUB operation instead of widening in the second iteration avoids the error. Thus, we add iteration count 2 to the set of hints.

Using the updated hints, we recompute the abstract fixpoint, taking care to use LUB operator after the second iteration. This results in the set of states $S'_2 \stackrel{\Delta}{=} (x \geq 0 \wedge m = 0) \vee (x \leq N \wedge x = m{+}1 \wedge m \geq 0)$ after the second iteration. Continuing the fixpoint computation, we apply widening after the third iteration resulting in a set of states $S'_3 \stackrel{\Delta}{=} (x \geq 0 \wedge m = 0) \vee (x \geq m{+}1 \wedge x \leq N \wedge m \geq 0)$. It turns out that S'_3 is a fixpoint for the loop. Further, it is strong enough to prove the assertion. Note that the computed loop invariant has a disjunction, and our refinement algorithm was necessary to prove the assertion.

The above description of our technique is informal and simplistic. We give a precise description in Section 2. In our second example, we have assumed the existence of widening operators that operate over finite powerset domains. Section 3 shows how to lift widening operators over base domains to widening operators over power-set domains, using the theory developed in [1]. In particular, we define a new connector $\boxplus$, which provides a lifted widening operator over the powerset domain with appropriate precision necessary for our purposes.

Widening is non-monotonic, and thus refining the widening operator in the earlier stages of fixpoint computation, could result in a larger set of states in a later iteration! We present a simple technique to avoid this problem in Section 4, using reachable states computed from the previous iteration.

We have implemented our technique for affine transition systems. Our implementation is able to prove invariants generated by the StInG tool [19] without doing any specialized analysis for linear relations. Section 5 presents empirical results from running our implementation. Our technique is independent of the specific abstract domain used. To illustrate this, Section 6 defines LUB and widen operators for an abstract domain of shape graphs, and enables our counterexample driven refinement to do shape analysis. Section 7 surveys related work and Section 8 concludes the paper.

2 Algorithm

We first present the algorithm in a very simple setting. Assume that we have a possibly infinite domain called **States**. We assume that the domain **States** has a precise LUB operator $\cup$, and a widening operator ∇. A transition system Θ is a pair $\langle I, \theta \rangle$. $I \subseteq \mathbf{States}$ and $\theta : 2^{\mathbf{States}} \rightarrow 2^{\mathbf{States}}$. Informally, θ is referred to as the "image" operator, which takes a set of current states as input, gives the set of possible next states as output. We use θ^{-1} to denote the "pre-image" operator, which takes the set of current states as input, and gives the set of previous states as output.

Transition systems are generated by *programs*. We describe the link between programs and transition systems below. A program P is a triple $\langle V, I, T\rangle$ where

- V is a finite set of variables, each of which takes valuations from a potentially infinite domain. A *state* is a valuation to all the variables in V. The set of all possible valuations to V is the domain **States**.
- I is a set of initial valuations to variables in V.
- $T \subseteq \textbf{States} \times \textbf{States}$ is a binary relation such that $T(s, s')$ holds whenever it is possible for the program to transition from state s to state s' in one step.

A program $P = \langle V, I, T\rangle$ gives rise to a transition system $\Theta = \langle I, \theta\rangle$, where $\theta(S) = \{s' \mid \exists s \in S.T(s, s')\}$, and $\theta^{-1}(S) = \{s \mid \exists s' \in S.T(s, s')\}$

A specification $\psi \subseteq \textbf{States}$ is a set of bad states that we do not want the system to reach. To check if a system $\Theta = \langle I, \theta\rangle$ satisfies a specification ψ, we first compute an over-approximation to the set of reachable states of the system, and check if the over-approximation intersects ψ. The least fixpoint $\textbf{PreciseReach}(\Theta) = \mu X.I \cup X \cup \theta(X)$ precisely represents the set of all reachable states of the system, though the fixpoint computation may not terminate. The system Θ satisfies specification ψ iff $\textbf{PreciseReach}(\Theta) \cap \psi = \emptyset$.

Widening operators from the abstract-interpretation community can help ensure termination of the fixpoint computation, at the cost of losing precision. If S_1 and S_2 are two sets such that $S_1 \subseteq S_2$, then $S_3 = S_1 \nabla S_2$ is a set such that $S_1 \subseteq S_3$ and $S_2 \subseteq S_3$. Further, there is some metric (such as the number of conjuncts in the formula representing the set) that decreases from S_1 to S_3. Thus, if we consider the least fixpoint $\textbf{WidenReach}(\Theta) = \mu X.(I \cup X)\nabla(I \cup X \cup \theta(X))$, it is guaranteed that (1) the computation of $\textbf{WidenReach}(\Theta)$ will terminate, and (2) $\textbf{WidenReach}(\Theta) \supseteq \textbf{PreciseReach}(\Theta)$. Thus, we can conclude that Θ satisfies specification ψ if $\textbf{WidenReach}(\Theta) \cap \psi = \emptyset$. On the other hand, if $\textbf{WidenReach}(\Theta) \cap \psi \neq \emptyset$, we cannot distinguish between the possibilities that either the system Θ does not satisfy ψ or the computation of **WidenReach** lost too much precision.

If we can keep track of the intermediate states in the fixpoint computation, then we can generate an *abstract counterexample* that can be automatically analyzed to classify if the error found is a false error or true error. If it is a false error, the analysis can also identify the precise point at which the abstract counterexample needs to be refined to avoid the recurrence of this specific false error.

More formally, let us consider the stages of the fixpoint computation $\textbf{WidenReach}(\Theta) = \mu X.(I \cup X)\nabla(I \cup X \cup \theta(X))$. Let $R_0 = I$, and let $R_i = R_{i-1}\nabla(R_{i-1} \cup \theta(R_{i-1}))$. Suppose n is the smallest index such that $R_n \cap \psi \neq \emptyset$. Let $\psi_n = \psi$. If $(R_{n-1} \cup \theta(R_{n-1})) \cap \psi_n = \emptyset$, then we note that step n is the exact index where the precision loss for the false error happened, and replace the widening operator with the LUB operator in that particular step of the fixpoint computation. Otherwise, if $(R_{n-1} \cup \theta(R_{n-1})) \cap \psi_n \neq \emptyset$, then we compute $\psi_{n-1} = \theta^{-1}(R_n \cap \psi_n)$, and check if $(R_{n-2} \cup \theta(R_{n-2})) \cap \psi_{n-1} = \emptyset$. This process continues until either an index is found where the widening operator needs to be refined into a LUB operator ($\cup$) to avoid the false error, or we find

```
AbsRefine(Θ = ⟨I, θ⟩, ψ)
  hints := ∅
  while true do
    (R,i,result) := AbstractFixPoint(Θ, ψ, hints)
    if result = true then
      return true
    else
      newHints := Refine(Θ, ψ, i, R)
      hints := hints ∪ {newHints}
    end if
  end while

Refine(Θ = ⟨I, θ⟩, ψ, count, R)
Requires count ≥ 0 ∧ R[count] ∩ ψ ≠ ∅
Returns step i where ∇ is replaced by ∪
  i := count
  while i > 0 do
    ψ := R[i] ∩ ψ
    if (R[i − 1] ∪ θ(R[i − 1])) ∩ ψ = ∅ then
      return i
    else
      i := i − 1; ψ := θ^{-1}(ψ)
    end if
  end while
  assert i = 0 ∧ R[i] ∩ ψ ≠ ∅
  print error trace and exit

AbstractFixPoint(Θ = ⟨I, θ⟩, ψ, hints)
Returns (R, i, result), where array R is
an array of set of states, result is boolean
and i is integer
  i := 1; R[0] := I
  if I ∩ ψ ≠ ∅ then
   return (R, 0,false)
  end if
  while true do
   if i ∈ hints then
     {precise next set of states}
     R[i] := R[i − 1] ∪ θ(R[i − 1])
   else
     {next set of states using widen}
     R[i] := R[i − 1]∇(R[i − 1] ∪ θ(R[i − 1]))
   end if
   if R[i] ∩ ψ ≠ ∅ then
     {We are not sure if System Θ satisfies ψ}
     return (R, i,false)
   end if
   if R[i] = R[i − 1] then
     {fixpoint, System Θ satisfies ψ}
     return (R, i,true)
   end if
   i := i + 1
  endwhile
```

Fig. 2. Iterative Refinement

that the repeated backward propagation of the error state intersects with the initial states $R_0 = I$, in which case we have evidence of a true error.

The procedure AbsRefine in Figure 2, together with the procedures AbstractFixPoint and Refine give a complete description of our iterative refinement procedure.

3 A Widening Operator for Finite Powerset Domains

In procedure AbstractFixPoint, we use a widening operator ∇ which takes as operands two arbitrary sets of states. The widening operator for convex polyhedral domain is studied at length in [9, 3]. The original widening operator as defined by Cousot and Halbwachs [9] did not allow for disjunctions in the first operand. If our abstraction refinement procedures are to be applied to powerset domains, a widening operator needs to be defined for that domain. In this section we show how to lift a widening operator over a base domain to a widening operator over its powerset domain. We follow the framework provided by Bagnara, Hill and Zaffanella [1], but define a new connector to provide the appropriate precision.

An abstract domain $\hat{D} = \langle D, \vdash, \mathbf{0}, \oplus \rangle$ is a join-semilattice where $\vdash$ is the partial order, $\mathbf{0}$ is the bottom element of the lattice and the LUB $d_1 \oplus d_2$ exists

for all $d_1, d_2 \in D$. For example, convex hull is such an operator for convex polyhedra. For all $d_1, d_2 \in D$, we will use the notation $d_1 \Vdash d_2$ to mean that $d_1 \vdash d_2$ and $d_1 \neq d_2$.

Let $d_1 = \wedge_i c_i$. Then, the standard widening operator [9] is defined as

$$d_1 \nabla d_2 \stackrel{\Delta}{=} \wedge\{c_j |\ d_2 \vdash c_j\}$$

For a set S, let $\wp(S)$ be the powerset of S, and let $\wp_f(S)$ be the set of all finite subsets of S. The operator $\oplus$ is overloaded so that, for each $S \in \wp_f(D)$, $\oplus S$ denotes the LUB of S. A set $S \in \wp(D)$ is *non-redundant* if and only if $\mathbf{0} \notin S$ and $\forall d_1, d_2 \in S : d_1 \vdash d_2 \Rightarrow d_1 = d_2$. The set of finite non-redundant subsets of D is denoted by $\wp_{fn}(D, \vdash)$. The reduction function $\Omega^{\vdash}_D : \wp_f(D) \rightarrow \wp_{fn}(D)$ maps each finite set into its non-redundant counterpart as follows:

$$\Omega^{\vdash}_D(S) \stackrel{\Delta}{=} S \setminus \{d \in S \mid d = \mathbf{0} \vee \exists d' \in S. d \Vdash d'\}$$

The finite powerset domain over $\hat{D}$ is the join-semilattice

$$\hat{D}_P = \langle \wp_{fn}(D, \vdash), \vdash_P, \mathbf{0}_P, \oplus_P \rangle$$

where $\mathbf{0}_P = \emptyset$ and $\forall S_1, S_2 \in \wp_{fn}(D, \vdash)$, $S_1 \oplus_P S_2 \stackrel{\Delta}{=} \Omega^{\vdash}_D(S_1 \cup S_2)$, and $S_1 \vdash_P S_2$ if and only if $\forall d_1 \in S_1 : \exists d_2 \in S_2. d_1 \vdash d_2$.

We say that $S_1 \preceq S_2$ if and only if either $S_1 = \mathbf{0}_P$ or $S_1 \vdash_P S_2$ and $\forall d_2 \in S_2 : \exists d_1 \in S_1. d_1 \vdash d_2$. Our goal is to define a *connector* operator $\boxplus$, such that for all $S_1, S_2 \in \wp_{fn}(D, \vdash)$, if $S_1 \vdash_p S_2$, then $S_1 \preceq (S_1 \boxplus S_2)$. Intuitively, $S_1 \boxplus S_2$ is obtained by minimally combining the elements of S_2 so as to obtain an S'_2 such that $S_1 \preceq S'_2$. More precisely, let $\hat{S}_2$ be a maximal subset of S_2 such that $\forall \hat{d} \in \hat{S}_2 : \exists d_1 \in S_1.\ d_1 \vdash \hat{d}.$ and let $\tilde{S}_2 \stackrel{\Delta}{=} \oplus\{d \mid d \in S_2 \setminus \hat{S}_2\}$. For any $\hat{d} \in \hat{S}_2$, let $J_{\hat{d}} \stackrel{\Delta}{=} (S_2 \setminus \{\hat{d}\}) \ \cup \ (\hat{d} \oplus \tilde{S}_2)$. We define $S_1 \boxplus S_2$ to be a minimal element(with respect to $\vdash_P$) from the set $\{J_{\hat{d}} \mid \hat{d} \in \hat{S}_2\}$. We find that this particular definition of the connector yields very good results in our abstraction refinement algorithm. It is easily checked that if $S_1 \vdash_p S_2$, then $S_1 \preceq S_1 \boxplus S_2$. Let $S_1, S_2 \in \wp_{fn}(D, \vdash)$, where $S_1 \Vdash S_2$. Then, $S_1 \nabla_P S_2$ is defined as follows:

$$\begin{array}{l} S_1 \nabla_P S_2 \stackrel{\Delta}{=} \ let \ \ S'_2 = \ if \ (S_1 \preceq S_2) \ then \ S_2 \ else \ S_1 \boxplus S_2 \ \ in \\ \qquad\qquad S'_2 \ \oplus_P \ \Omega^{\vdash}_D(\{d_1 \nabla d_2 \in D \mid d_1 \in S_1, d_2 \in S'_2, d_1 \Vdash d_2\}) \end{array}$$

To illustrate the need for the connector operator, recall the example from Figure 1(b) from Section 1. Recall that during the second iteration of the refinement loop, the symbolic state after second iteration of the fixpoint is $S'_2 \stackrel{\Delta}{=} (x \geq 0 \ \wedge \ m = 0) \vee (x \leq N \ \wedge \ x = m + 1 \ \wedge \ m \geq 0)$. The new symbolic state that is generated after one more execution of the loop body is $S_{new} = (x \leq N \wedge x = m+2 \wedge m \geq 0)$. Here $S''_2 \stackrel{\Delta}{=} S'_2 \oplus_P S_{new} \stackrel{\Delta}{=} (x \geq 0 \wedge m = 0) \vee (x \leq N \ \wedge \ x = m + 1 \ \wedge \ m \geq 0) \vee (x \leq N \ \wedge \ x = m + 2 \ \wedge \ m \geq 0)$. Our goal is to compute $S'_3 \stackrel{\Delta}{=} S'_2 \ \nabla_P \ S''_2$. Since $S'_2 \preceq S''_2$ does not hold, we need to compute $S'_2 \boxplus S''_2$ by merging some of the elements of S''_2. Here

$\hat{S_2''} \stackrel{\Delta}{=} (x \geq 0 \ \wedge \ m = 0) \vee (x \leq N \ \wedge \ x = m+1 \ \wedge \ m \geq 0)$ and $\tilde{S_2''} \stackrel{\Delta}{=} (x \leq N \ \wedge \ x = m+2 \ \wedge \ m \geq 0)$. If we merge $\tilde{S_2''}$ with first element of $\hat{S_2''}$ then the result is $S_2' \boxplus S_2'' \stackrel{\Delta}{=} (x \geq 0 \ \wedge \ m \geq 0 \ \wedge \ x \geq m)$ whereas if we merge $\tilde{S_2''}$ with second element of $\hat{S_2''}$ then the result is $S_2' \boxplus S_2'' \stackrel{\Delta}{=} (x \geq 0 \wedge m = 0) \vee (x \leq N \ \wedge \ m \geq 0 \ \wedge \ x \geq m+1 \ \wedge \ x \leq m+2)$. The result of widening in the first case is $x \geq 0 \ \wedge \ m \geq 0$ which is less precise than the widening result of second case $(x \geq 0 \ \wedge \ m = 0) \vee (x \leq N \ \wedge \ x \geq m+1 \ \wedge \ m \geq 0)$ when using the widening operator defined in [9] for the base domain of convex polyhedra. Thus, it is seen that choosing the minimal result (second case) is necessary to obtain a fixpoint that is strong enough to prove the assertion. Our definition of connector is necessary to prove this example, and most of the other examples we have encountered.

Using the theory developed in [1], it can be shown that ∇_P satisfies the convergence properties of a widening operator. Without transforming S_2 to S_2' using the $\boxplus$ connector, such convergence guarantees cannot be given (see [1]). In Section 2, when we discuss the algorithm, we did not explicitly mention the difficulties of dealing with powerset domains. The operator $\cup$ used in Section 2 corresponds to $\oplus_P$ operator defined in this Section.

4 Dealing with Non-monotonicity

One technical issue with widening is its non-monotonicity. That is, if $S_1 \subseteq S_1'$ and $S_2 \subseteq S_2'$, then it is not necessarily the case that $(S_1 \nabla S_2) \subseteq (S_1' \nabla S_2')$. Thus, refining the widening operator to a least upper bound operation in step i of the abstract fixpoint computation, could result in a larger set of states in a later iteration! However, this problem can be easily avoided since we already keep track of the intermediate set of states reached at each iteration of the abstract fixpoint computation. At every iteration of the abstract fixpoint computation, we can intersect the states reached at step i with the set of states reached at step i during the previous iteration of the abstract fixpoint computation. If the step count i is greater than the number of steps required in the previous iteration, then we can intersect with the fixpoint computed in previous iteration. The modified algorithm is shown in [11].

Progress guarantee. With the monotonic abstraction refinement procedure, it is clear that in the successive abstraction iterations we compute more precise abstract fixpoint as compared to the previous iteration. We can make a stronger statement about progress by defining an ordering between counterexamples. An abstract counterexample C is a sequence of set of states $R_0, R_1, \ldots, R_n$ such that (1) R_i is the set of states computed in step i of the abstract fixpoint computation, and (2) $R_n \cap \psi \neq \emptyset \ \wedge \ \forall i < n. R_i \cap \psi = \emptyset$. The length of counterexample C is denoted by $|C|$. We define a binary relation $\prec_c$ on abstract counterexamples as $C_1 \prec_c C_2$ iff either (1) $|C_1| < |C_2|$, or (2) $|C_1| = |C_2|$ and $\forall i.0 \leq i < |C_1| : R_i(C_2) \subseteq R_i(C_1) \ \wedge \ \exists i.0 \leq i < |C_1| : R_i(C_2) \subset R_i(C_1)$. We state our progress guarantees below.

Theorem 1. *Let C_i be the abstract counterexample generated during the iteration i of abstraction computation. Then, we have for all $i \geq 0$, $C_i \prec_c C_{i+1}$.*

Lemma 1. *Let C_i be the counterexample of length n generated during the ith iteration of abstraction, then after at most n iterations of refinement and abstraction, counter examples generated, if any, will be of length greater than n.*

The proof of Theorem 1 is given in [11]. Lemma 1 follows from Theorem 1 and the fact that the set of hints monotonically increases in successive iterations of refinement. All these results make use of our assumption from Section 2 that the LUB operator $\cup$ in algorithm MAFixpoint is precise. Lemma 1 guarantees that if an abstract counterexample is a false error, then it will necessarily get refined in bounded number of refinement iterations and will never reappear as an abstract counterexample at subsequent iterations of iterative refinement. However, there is no guarantee that the iterative refinement loop will ever terminate. In practice, we terminate the outer iterative refinement loop after a certain time or memory limit is exhausted and return the answer "don't know".

Systematic abstraction refinement. For powerset abstract domains like the sets of convex polyhedra, the least upper bound operator $\oplus$ is the non-redundant union as defined in Section 3. Thus the refinement will add more and more disjuncts to the reachable set of states. It is possible that this increase in the number of disjuncts will continue infinitely even though the assertion can be satisfied by an abstract fixpoint computed by merging some intermediate disjuncts and doing widening later on. Thus intermediate merging may provide convergence. We use the *connector* $\boxplus$ operator described in Section 3 as an operator to merge some disjuncts into one convex polyhedra. The refinement algorithm now checks whether using the merging operation instead of widening avoids error. If it does then widening operator is replaced by the merge operation. Thus we have three upper bound operators $\cup$, $\boxplus$ and ∇ of decreasing precision. The refinement algorithm can now refine a widening operator to either $\boxplus$ operator or a $\cup$ operator. It can also refine the $\boxplus$ operator to $\cup$ operator. The hints that are generated by the refinement algorithm now are of the form $\langle i, op \rangle$, where i is the step number and $op \in \{\cup, \boxplus\}$ is the operator to be applied after that step. The procedure $MAFixPoint$ in [11] gives the abstraction procedure which ensures monotonicity and uses the new hints just described.

If the refinement algorithm returns $\langle i, \cup \rangle$ then, it is clear that refining the widening operator in step i to LUB will remove the abstract counterexample. However, if the refinement algorithm returns $\langle i, \boxplus \rangle$, then the widening operator in step i could be replaced either by a $\cup$ or by $\boxplus$. It is not clear whether it is provident to convert the widening operation to $\cup$ or $\boxplus$ in this case. It is possible that exactly one choice results in computation of the abstract fixpoint necessary to prove the property, whereas the other choice leads to non-termination of the abstraction refinement cycle. Thus, it is more advantageous to try both possibilities. The procedure SARefinement in [11] systematically tries both possibilities if the refinement returns $\langle i, \boxplus \rangle$.

5 Implementation and Empirical Results

Our implementation is for an imperative language with integer variables, and usual control structures including sequencing, conditionals and loops. The implementation is based on the algorithm SARefinement from [11], but it differs in three ways: (1) While the algorithm SARefinement from [11] is fully symbolic, the implementation does a mixture of explicit and symbolic state exploration. In particular, the control (program counter) is kept explicit, and we never merge symbolic constraints from two different program counters. (2) Widening is performed in our implementation only along the back edges of loops. Thus, at join points such as the end of if-then-else statements, the states from the two branches are kept separate unless they turn out to be identical. (3) We use two heuristics to prune the space of refinements explored by the algorithm. The first heuristic disallows conversion of widening to $\boxplus$ in consecutive iterations of refinement. In such a case, we convert widening to $\cup$ in the latter iteration. The second heuristic restricts the size of set H in procedure SARefinement to 3. Any successive refinement does not increase size of H, but converts the widening to one of $\boxplus$ or $\cup$ depending on the first heuristic.

Our prototype implementation uses the library PPL [2] for polyhedral operations. We have run our program on a machine with Intel Pentium 3.0GHz processor, 512 KB cache and 512 MB RAM.

We have experimented with two widening operators for polyhedral domain: (1) the original widening operator defined by Cousot and Halbwachs [9] (we refer to this as "CH78"), and (2) the widening operator defined by Bagnara, Hill, Ricci and Zaffanella [3] (we refer to this as "BHRZ03"). Both these widening operators are defined for convex polyhedra. Since our abstract domain is the set of convex polyhedra, we use Section 3 to lift these widening operators to the powerset domain.

We evaluated the implementation on two sets of examples. The results show that our technique is robust, regardless of the widening operator used.

The first set of examples were obtained from Rustan Leino [15], and are part of his test suite for the tool Boogie. All these programs have embedded assertions in the program and our goal is to discover loop invariants strong enough to prove the assertions. Two of these programs (Prog 7 and Prog 9) are incorrect, in that the assertions fail. Table 1 shows for each of the two different widening operators, the time taken, and the number of widenings that were converted to $\cup$ and $\boxplus$ respectively, to either prove the assertion or find the error. Our implementation is able to successfully prove the assertions in all the 11 correct programs, and it is able to find the error in the two erroneous programs. Out of the 11 correct programs, 5 programs require non-trivial iterative refinement. Prog0 is the example from Figure 1(b). Prog8 is very similar to Prog0, except that the assertion after the while loop is stronger. It asserts that $(0 \leq m < N \ \wedge \ x = N)$. Our iterative refinement computes the more precise invariant $(x - m = 1 \ \wedge \ 0 \leq m \leq 1 \ \wedge \ 1 + m \leq N) \vee (m = 0 \ \wedge \ 0 \leq x \leq 1) \vee (m = 0 \ \wedge \ 1 \leq x \leq N) \vee (2 \leq x \leq N \ \wedge \ m \geq 0 \ \wedge \ x \geq m + 1)$ needed to prove the property. The choice of the widening operator (CH78 or BHRZ03) influences only the number of refinement steps, but not the ability of our technique to prove the property.

Table 1. Experimental results, Programs on left are from Rustan Leino [15], Programs on right are from the StInG web page [18]. The column head (I) indicates the time (in sec) taken for example program to be verified and column head (II). indicates the number of refinement steps ($\cup, \boxplus$). * programs are incorrect(i.e., assertion fails).

Program	CH78		BHRZ03		Program	CH78		BHRZ03	
Name	I	II	I	II	Name	I	II	I	II
Prog0	0.055	(1, 1)	0.054	(1, 1)	See-Saw	0.816	(3, 3)	0.811	(3, 3)
Prog1	0.01	(0, 0)	0.011	(0, 0)	Robot-HH96	0.01	(0, 0)	0.01	(0, 0)
Prog2	0.012	(0, 0)	0.014	(0, 0)	Berkeley	0.098	(1, 1)	0.085	(0, 1)
Prog3	0.014	(0, 0)	0.01	(0, 0)	Berkeley-nat	0.432	(2, 1)	3.44	(2, 1)
Prog4	0.047	(2, 0)	0.138	(2, 2)	Heapsort	0.719	(2, 1)	0.162	(0, 1)
Prog5	0.058	(2, 0)	0.093	(2, 1)	Train-RM03	0.022	(0, 0)	0.02	(0, 0)
Prog6	0.035	(1, 0)	0.021	(0, 0)	EFM	0.06	(0, 0)	0.06	(0, 0)
Prog7*	0.01	(0, 0)	0.009	(0, 0)	EFM1	0.06	(0, 0)	0.06	(0, 0)
Prog8	0.097	(2, 1)	0.268	(3, 2)	LIFO	3.325	(3, 3)	1.29	(2, 1)
Prog9*	0.01	(0, 0)	0.008	(0, 0)	LIFO-NAT	29.55	(7, 5)	2.537	(2, 2)
Prog10	0.015	(0, 0)	0.011	(0, 0)	cars-midpt	$\geq$10000	($\geq$3, $\geq$3)	$\geq$10000	($\geq$3, $\geq$3)
Prog11	0.029	(0, 0)	0.032	(0, 0)	barber	10.48	(3, 3)	17.12	(3, 3)
Prog12	0.01	(1, 0)	0.014	(1, 0)	Swim-pool	11.13	(3, 3)	18.029	(3, 3)
					Swim-pool-1	11.24	(3, 3)	18.50	(3, 3)

The second set of examples are available at the StInG website [18]. The StInG tool [18, 19] uses Farkas' Lemma to synthesize the strongest linear invariant. We requested Sriram Sankaranarayanan to provide the exact invariants that StInG computes. Then, we modified the examples to assert the invariant computed by StInG (the invariants are also now available at [18]). Then, we used our iterative refinement algorithm to prove these assertions. Our implementation is able to prove the invariants in all examples with the exception of the program 'cars-midpt'. This run took more than 10000 seconds but had done only 6 refinement steps. Thus, we are unsure if the algorithm will converge if more refinement steps can be executed. We find that the polyhedral operations are very time consuming in this example. The last powerset widening call to the PPL library took 6910 seconds, where both arguments to this widening operation had 4 disjuncts with an average of 125 constraints per disjunct.

Though the invariants generated by StInG itself do not contain disjunctions, our iterative refinement algorithm used disjunctions in some cases to prove these invariants. Again, the choice of the widening operator (CH78 or BHRZ03) influences only the number of refinement steps, but not the ability of our technique to prove the property.

6 Shape Analysis

We briefly sketch an abstract domain for representing heap configurations, with LUB and widen operators. This immediately enables application of our iterative refinement algorithm to do shape analysis [17]. Our approach does not capture

reachability information in the heap, and only certain programs with unbounded state spaces can be verified using our approach. Our approach currently does not have all the sophistications of [17].

A *Boolean Linked List Program* (BLL Program for short) is a single-procedure program with a finite number of variables, where every variable has the following datatype:

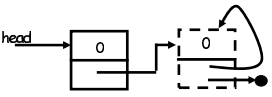
```
class Node {
  bool data;
  Node next;
}
```

BLL programs allow dynamic creation of objects, so they have potentially infinite state spaces. Our abstract domain is the domain of abstract heap graphs $\hat{\mathcal{G}} = \langle \mathcal{G}, \vdash, \mathbf{0}_{\mathcal{G}}, \oplus \rangle$. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the variables in a BLL Program P. An abstract heap graph A of program P is a 5-tuple $\langle \mathbf{U^A}, \mathbf{V^A}, \mathbf{Data^A}, \mathbf{Next^A}, \mathbf{Z^A} \rangle$, where (1) $\mathbf{U^A}$ is a finite set of nodes $\{U_0, U_1, \ldots, U_k\}$, (2) $\mathbf{V^A}$: $V \rightarrow 2^{\mathbf{U^A}}$, maps every variable to a set of nodes, (3) $\mathbf{Data^A} : \mathbf{U^A} \rightarrow \{0, 1, \top\}$ maps the `data` field of each node to boolean values or $\top$, and (4) $\mathbf{Next^A}$: $\mathbf{U^A} \rightarrow 2^{\mathbf{U^A}}$, maps the `next` field of each node to a set of nodes, and (5) $\mathbf{Z^A} \subseteq \mathbf{U^A}$ is the subset of nodes that are designated as summary nodes. An abstract heap graph represents a set of concrete heap graphs. A concrete heap graph represents a state of a BLL Program, which is a set of heap addresses, with variables pointing to specific addresses, and specific concrete values to objects in each of these addresses. The concretization function γ maps every abstract heap graph to a set of concrete heap graphs. The LUB of two abstract heap graphs A and B such that $\mathbf{U^A} \cap \mathbf{U^B} = \emptyset$, is intuitively just the disjoint union of the two heap graphs. The widen of two abstract heap graphs A and B, given by $C = A \nabla B$ is intuitively obtained by fixing the nodes of the result to $\mathbf{U^A}$ and adding more edges representing B into A, and updating to coarser data values representing nodes of B into nodes of A. A precise description of the concretization function γ, the LUB operator, and the widening operator can be found in [11].

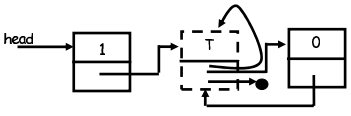
The intuition is that the application of the LUB operator can add more nodes to the abstract heap graph, but the application of the widening operator cannot. Thus, if the LUB operator is used only a finite number of times during the fixpoint, the number of nodes in the abstract heap graph stops growing after a finite number of iterations. Thereafter, the fixpoint converges after sufficient applications of widening.

Using this abstract domain, we are able to prove some programs that allocate unbounded number of nodes by using our refinement algorithm. Consider the program shown in Figure 3. First, our abstract fixpoint computation uses widening along the back edge of every while loop. Intuitively, repeated applications of the widening operator allow only one summary node, which results in the second abstract heap graph in Figure 3. This invariant is not strong enough to prove the assertion. Then, our refinement algorithm detects that the second widening in the second while loop is the reason for the loss of precision and

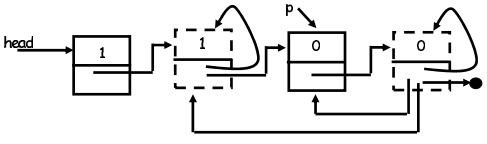
```
head := null; p := null;
while (*) {
    p := new node;
    p.data := 0;
    p.next := head;
    head := p;
}
p := head;
while (p != null) {
    assume (p != null);
    p.data = 1;
    p := p.next;
}
assume (p = null);
p := head;
while (*) {
    assume (p != null);
    assert (p.data = 1);
    p := p.next;
}
```



Abstract heap graph after first while loop



Imprecise heap graph obtained using widening in the second while loop (not sufficient to prove assertion)



Heap graph obtained after doing one refinement, where widening in the second iteration of the second while loop is converted to LUB (sufficient to prove assertion)

Fig. 3. Example program that creates an unbounded linked list

converts this widen to LUB. After this refinement, subsequent applications of the widening operator are now able to allow two summary nodes as shown in the third abstract heap graph in Figure 3. In this domain, converting widen to LUB results in allowing more nodes in the the resulting fixpoint. This is analogous to adding more disjuncts by converting widen to LUB in the powerset domain of convex polyhedra.

7 Related Work

Counterexample driven refinement [14], has gained popularity in recent years [7, 4], as a technique to prove properties of systems, while reducing false errors. Several tools based on counterexample driven refinement have appeared in the past few years [4, 12, 6]. All these efforts use predicate abstraction [10], which is a particular case of abstract interpretation [8]. In contrast to these efforts, we present a technique to refine any abstract interpretation automatically using counterexamples. Thus, our work has the potential to make counterexample driven refinement more broadly applicable to a variety of abstract domains, in order to reduce false errors.

Techniques to reduce precision loss due to widening have been studied in the abstract interpretation community. We compare our work with (1) generic

approaches that work for any abstract domain, and (2) specific approaches for particular abstract domains. In the category of generic approaches, Jeannet, Halbwachs and Raymond partition the abstract domain with predicates on the control state [13] to improve precision. They first perform a combination of forward and backward analysis, and use predicates present in the conditionals in the program to do such partitioning. Unlike their approach we use a backward propagation of the abstract counterexample in the spirit of [7] to generate refinement hints. Bourdouncle has noted that more precise abstract domains can be obtained by applying widening only in certain equations (cutting of dependence loops) [5]. Bourdouncle also defines new widening operators together with disjunctive completion by representing sets of abstract elements. This approach does not use counterexamples to refine the abstract domain. In the category of approaches that are specific to particular domains, the StInG tool [18, 19] uses Farkas' Lemma to synthesize linear invariants by extracting non-linear constraints on the coefficients of a target invariant from an affine program. Unlike StInG, our technique uses fixpoints, and is independent of the abstract domain.

Leino and Logozzo use counterexample contexts obtained from the theorem prover to re-run the abstract interpreter restricted to the counterexample context, with the hope of obtaining more precise loop invariants [16]. This approach has philosophical similarities to our approach, but there are several technical differences. Their approach is implemented entirely inside the theorem prover, unlike ours. Unlike the technique presented here, there is no progress guarantee with their approach, and their technique does not stop the iterative refinement if there is a true error.

8 Conclusion

We presented a new counterexample driven refinement technique that can refine any abstract interpretation, and tune the precision depending on the property of interest. Our technique is independent of the abstract domain used. We instantiated the technique for affine programs and our implementation is able to prove invariants generated by the StInG tool. We also sketched how the technique can be applied to do shape analysis.

Acknowledgment. We thank Rustan Leino for providing his example programs from the Boogie project, and Sriram Sankaranarayanan for providing us the invariants generated by StInG. We thank Supratik Chakraborty, Prasad Naldurg and Mooly Sagiv for insightful discussions.

References

1. R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. In *VMCAI 04: Verification, Model Checking and Abstract Interpretation*. Springer-Verlag, 2004.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. PPL: The Parma Polyhedral Library — http://www.cs.unipr.it/ppl/.

3. R. Bagnara, P.M. Hill, E. Ricci, and E. Zaffanella. Precise widening opertors for convex polyhedra. In *SAS 03: Static Analysis*. Springer-Verlag, 2003.
4. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057. Springer-Verlag, 2001.
5. F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–423, 1992.
6. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer-Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
9. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78: Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
10. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
11. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. Technical Report MSR-TR-2006-02, Microsoft Research, 2006.
12. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02*, pages 58–70. ACM, January 2002.
13. B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *SAS 99: Static Analysis*, LNCS 1694, pages 39–50. Springer-Verlag, 1999.
14. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
15. K. Rustan M. Leino. Personal communication, September 2005.
16. K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS 2005: Asian Symposium on Programming Languages and Systems*, 2005. To appear.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.
18. S. Sankaranarayanan. StInG: The Stanford Invarint Generator — http://theory.stanford.edu/ srirams/software/sting.html.
19. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint based linear-relations analysis. In *SAS 04: Static Analysis*. Springer-Verlag, 2004.