Craig Interpretation

Aws Albarghouthi[1], Arie Gurfinkel[2], and Marsha Chechik[1]

[1]Department of Computer Science, University of Toronto, Canada
[2]Software Engineering Institute, Carnegie Mellon University, USA

Abstract. Abstract interpretation (AI) is one of the most scalable automated approaches to program verification available today. To achieve efficiency, many steps of the analysis, e.g., joins and widening, lose precision. As a result, AI often produces false alarms, coming from the inability to find a safe inductive invariant even when it exists in a chosen abstract domain.

To tackle this problem, we present VINTA, an iterative algorithm that uses *Craig interpolants* to refine and guide AI away from false alarms. VINTA is based on a novel refinement strategy that capitalizes on recent advances in SMT and interpolation-based verification to (a) find counterexamples to justify alarms produced by AI, and (b) to strengthen an invariant to exclude alarms that cannot be justified. The refinement process continues until either a safe inductive invariant is computed, a counterexample is found, or resources are exhausted. This strategy allows VINTA to recover precision lost in many AI steps, and even to compute inductive invariants that are inexpressible in the chosen abstract domain (e.g., by adding disjunctions and new terms).

We have implemented VINTA and compared it against top verification tools from the recent software verification competition. Our results show that VINTA outperforms state-of-the-art verification tools.

1 Introduction

Abstract interpretation (AI) is one of the most scalable automated approaches to program verification available today. AI iteratively computes an *inductive invariant* I of a given program P in a chosen abstract domain D. P is *safe*, i.e., it cannot reach an error location e, if I is *safe*, i.e., it does not include e. The price of AI's efficiency is false alarms (i.e., inability to find a safe I even when it exists in D), that are introduced through imprecision inherent in many steps of the analysis (e.g., join and widening).

In this paper, we present VINTA[1], an iterative algorithm that uses *Craig interpolants* [8] to refine and guide AI away from false alarms. VINTA marries the efficiency of AI with the precision of Bounded Model Checking (BMC) [6] and the ability to generalize from concrete executions of interpolation-based software verification [15, 1].

The main phases of the algorithm are shown in Fig. 1. Given a program P and a safety property φ, VINTA starts by computing an inductive invariant I of P using an abstract domain D (the AI phase). If I is safe (i.e., $I \Rightarrow \varphi$), then P is safe as well. Otherwise, VINTA goes to a novel refinement phase. First, refinement

[1] Verification with INTerpolation and Abstract interpretation.

uses BMC to check for a counterexample in the explored part of P. Second, if BMC fails to find a counterexample, it uses an interpolation-based procedure to strengthen I to I'. If I' is not inductive (checked in the "Is Inductive?" phase), the AI phase is repeated to weaken I' to include all reachable states of P. This process continues until either a safe inductive invariant or a counterexample is found, or resources (i.e., time or memory) are exhausted.

In our experience, VINTA is able to recover precision lost due to widening, join, imprecise post-image, and inexpressiveness of the chosen domain D. Furthermore, unless aborted, it never produces false alarms. While we present VINTA as a refinement strategy for AI, it can equivalently be seen as an interpolation-based verification algorithm guided by AI. Indeed, we show that both the BMC and interpolation phases benefit greatly from the invariants discovered by AI. We have implemented VINTA in UFO [2, 1], our software verification framework built on top of the LLVM compiler [14]. For evaluation, we used benchmarks from the recent Software Verification Competition (SV-COMP) [4]. We have compared several configurations of VINTA with our prior tool, UFO [1], and with the top two tools from SV-COMP, CPACHECKER-ABE and CPACHECKER-MEMO. The results show that VINTA outperforms these state-of-the-art approaches.

This paper makes several contributions. First, the AI phase is a novel AI-based invariant computation algorithm. It works on a summary of a Control Flow Graph (CFG) that contains only loop-heads. It efficiently maintains disjunctive loop invariants. Finally, it provides counterexamples to justify alarms. Second, we present a new widening strategy that extends widening from a given domain D to its finite powerset $\mathcal{P}_f(D)$. Third, we present a novel refinement strategy for strengthening invariants and eliminating potential false alarms. Unlike existing work on interpolation-based refinement (e.g., [1, 15]), our strategy is both guided and bounded by the invariants discovered by AI. Finally, we show empirically that the new approach outperforms other state-of-the-art techniques on a collection of software verification benchmarks.

Related Work. Our approach is closely related to the DAGGER tool of Gulavani et al. [10] that is also based on refining AI, and to our earlier tool UFO [1] that combines predicate abstraction with interpolation-based verification. The key differences between VINTA and DAGGER are: (1) DAGGER can only refine imprecision caused by widening and join. VINTA can refine imprecision up to the concrete semantics of the program (as modeled in SMT). (2) DAGGER refines joins explicitly, which may result in an exponential increase in the number of abstract states compared to the size of the program. VINTA refines joins implicitly using interpolants and SMT. (3) DAGGER requires a specialized interpolation procedure, which, so far, has only been developed for the octagon and the polyhedra domains. VINTA can use any off-the-shelf interpolating SMT solver, immediately benefiting from any advances in the field.

Compared to UFO, VINTA improves both the exploration algorithm (by extending it to an arbitrary abstract domain) and the refinement procedure (by extending it to use intermediate invariants computed by AI). Both of these

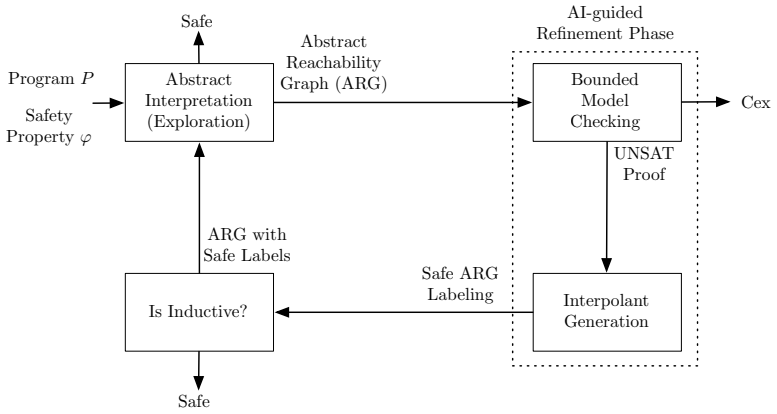

Fig. 1. High-level overview of VINTA.

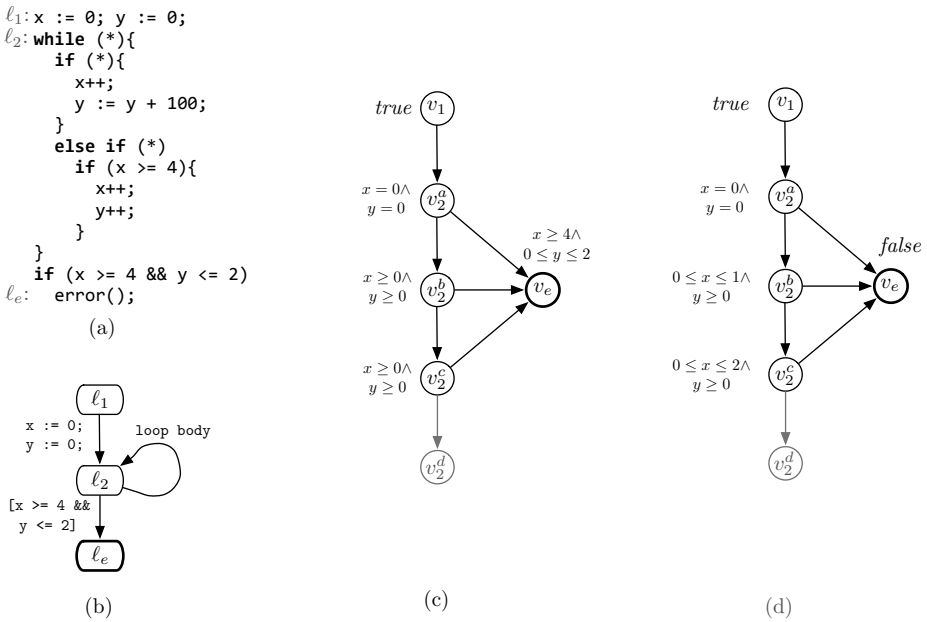

Fig. 2. (a) A safe program P ('$*$' denotes a nondeterministic choice). (b) A cutpoint graph of P. An ARG of P after (c) the first and (d) after the second AI step.

extensions are important for VINTA's success, as shown in the experiments in Sec. 5.

The rest of the paper is organized as follows: Sec. 2 gives a general overview of VINTA. Sec. 3 provides the notation and definitions required for the paper. Sec. 4 formally presents our algorithm. Sec. 5 describes our implementation, optimizations and experimental results. Finally, Sec. 6 concludes the paper.

2 Example

In this section, we illustrate VINTA on proving safety (i.e., unreachability of ℓ_e) of program P from [10], shown in Fig. 2(a). P is known to be hard to analyze without refinement, and even the refinement approaches of [9] and [17] fail to solve it (see [10] for details). DAGGER [10] (the state-of-the-art in AI-refinement) solves it using the domain of polyhedra by computing the safe invariant $x \leq y \leq 100x$. Here, we show how VINTA solves the problem using the BOX domain and refinement to compute an alternative safe invariant: $x \geq 4 \Rightarrow y > 100$. In this example, the refinement must recover imprecision lost due to widening and join, and extend the base-domain with disjunction. All of this is done automatically via an SMT-based interpolation procedure. Due to space limitations, we show only the first few iterations of the analysis.

Step 1.1: AI. VINTA works on a *cutpoint graph* (CPG) of a program: a collapsed CFG where the only nodes are cutpoints (loop-heads), entry, and error locations. A CPG for P is shown in Fig. 2(b).

VINTA uses a typical AI-computation following the *recursive iteration strategy* [7] and widening at every loop unrolling. Additionally, it records the finite traces explored by AI in an *Abstract Reachability Graph* (ARG). An ARG is an unrolling of the CPG. Each node u of an ARG corresponds to some node v of a CPG, and is labeled with an over-approximation of the set of states reachable at that point.

Fig. 2(c) shows the ARG from the first AI-computation on P. Each node v_i in the ARG refers to node ℓ_i in the CPG. The superscript in nodes v_2^a, v_2^b, v_2^c, and v_2^d is used to distinguish between the different unrollings of the loop at ℓ_2. The labels of the nodes v_2^a, v_2^b, and v_2^c over-approximate the states reachable before the first, second, and third iterations of the loop, respectively. The node v_2^c is said to be *covered* (i.e., subsumed) by $\{v_2^a, v_2^b\}$. The labels of the set $\{v_2^a, v_2^b\}$ form an inductive invariant $\mathcal{I}_1 \equiv (x \geq 0 \wedge y \geq 0)$. The node v_2^d is called an *unexplored child*, and has no label and no children. It is used later when AI-computation is restarted. Finally, note that $\mathcal{I}_1$ is not safe (the error location v_e is not labeled by *false*), and thus refinement is needed.

Step 1.2: AI-guided Refinement. First, VINTA uses a BMC-style technique [11] to check using an SMT-solver whether the current ARG has a feasible execution to the error node ℓ_e. There is no such execution in our example (see Fig. 2(c)) and the algorithm moves to the next phase.

The second phase of refinement is based on a novel interpolation-based procedure that is described in detail in Sec. 4. Specifically, the procedure takes the current ARG (Fig. 2(c)) and its labeling and produces a new safe (but not necessarily inductive) labeling shown in Fig. 2(d). Here, refinement reversed the effects of widening by restoring the upper bounds on x. Note that the new labels are stronger than the original ones – this is guaranteed by the procedure and the original labels are used to guide it.

Step 1.3: Is Inductive? The new ARG labeling (Fig. 2(d)) is not inductive since the label of v_2^c is not contained in the label of v_2^b (checked by an SMT-solver), and another AI phase is started.

Step 2.1: AI (again). AI is restarted "lazily" from the nodes that have unexplored children. Here, v_2^c is the only such node. This ensures that AI is restarted from the inner-most loop where the invariant is no longer inductive. First, the label of v_2^c is converted into an element of an abstract domain by a given abstraction function. In our example, the label is immediately expressible in Box, so this step is trivial. Then, AI-computation is restarted as usual.

In the following four iterations (omitted here), refinement works with the AI-based exploration to construct a safe inductive invariant $x \geq 4 \Rightarrow y > 100$. Note that since the invariant contains a disjunction, this means refinement had to recover from imprecision of join (as well as recovering from imprecision due to widening shown above).

This example is simple enough to be solved with other interpolation-based techniques, but they require more iterations. Ufo [1], our prior approach without AI-based exploration and refinement, needs nine iterations, and a version of Vinta with unguided refinement from Ufo needs seven. Our experiments suggest that this translates into a significant performance difference on bigger programs.

3 Definitions

In this section, we present the definitions and notation used in the rest of the paper.

Programs as Cutpoint Graphs. We represent a program by a *cutpoint graph* (CPG), a collapsed form of a CFG where each node is a loop-head and each edge is a loop-free path between two loop-heads. Formally, a *program* P is a tuple $(\mathcal{CP}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$, where $\mathcal{CP}$ is a finite set of cutpoints, δ is a finite set of actions, $\mathsf{en} \in \mathcal{CP}$ is a special cutpoint denoting the entry location of P, $\mathsf{err} \in \mathcal{CP}$ is the error cutpoint, and Var is the set of variables of program P. An *action* $(\ell_1, T, \ell_2) \in \delta$ represents loop-free paths between ℓ_1 and ℓ_2, where $\ell_1, \ell_2 \in \mathcal{CP}$ and T is the set of statements along the paths. We assume that there does not exist an action $(\ell_1, T, \ell_2) \in \delta$ s.t. $\ell_1 = \mathsf{err}$. T can be viewed as a transition relation over the variables $\mathsf{Var} \cup \mathsf{Var}'$, where Var' is the set of primed versions of variables in Var. We write $[\![T]\!]$ for the standard semantics of a statement T. For example, if T is `if x = 0 then x := 1 else x := 2`, then $[\![T]\!] \equiv (x = 0 \Rightarrow x' = 1) \wedge (x \neq 0 \Rightarrow x' = 2)$.

A program P is *safe* iff there does not exist an execution that starts in en and reaches err through the actions in δ.

Weak Topological Ordering. A *Weak Topological Ordering* (WTO) [7] of a directed graph $G = (V, E)$ is a well-parenthesized total-order, denoted $\prec$, of V without two consecutive "(" s.t. for every edge $(u, v) \in E$:

$$(u \prec v \wedge v \notin \omega(u)) \vee (v \preceq u \wedge v \in \omega(u)),$$

where elements between two matching parentheses are called a *(wto-)component*, the first element of a component is called a *head*, and $\omega(v)$ is the set of heads of all components containing v.

Let $v \in V$, and U be the innermost component that contains v in the WTO. We write $\textsc{WtoNext}(v)$ for an element $u \in U$ that immediately follows v, if it exists, and for the head of U otherwise.

Let U_v be a component with head v. First, suppose that U_v is a subcomponent of some component U. If there exists a $u \in U$ s.t. $u \notin U_v$ and u is the first element in the total-order s.t. $v \prec u$, then $\textsc{WtoExit}(v) = u$. Otherwise, $\textsc{WtoExit}(v) = w$, where w is the head of U. Second, suppose that U_v is not a subcomponent of any other component, then $\textsc{WtoExit}(v) = u$, where u is the first element in the total-order s.t. $u \notin U_v$ and $v \prec u$. Intuitively, if the WTO represented program locations, then $\textsc{WtoExit}(v)$ is the first control location visited after exiting the loop headed by v. For example, for the program in Fig. 2(b), a WTO of the control locations is $\ell_1(\ell_2)\ell_3$, where ℓ_2 is the head of the component comprising the while loop. $\textsc{WtoNext}(\ell_2) = \ell_2$ and $\textsc{WtoExit}(\ell_2) = \ell_3$. Note that WtoNext and WtoExit are partial functions and we only use them where they have been defined.

Abstract Reachability Graphs (ARGs). Let $P = (\mathcal{CP}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ be a program. An *Abstract Reachability Graph* (ARG) of P is a tuple $(V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$, where (V, E, v_{en}) is a directed acyclic graph (DAG) rooted at the *entry node* $v_{\mathsf{en}} \in V$, $\nu : V \to \mathcal{CP}$ is a map from nodes to cutpoints of P where $\nu(v_{\mathsf{en}}) = \mathsf{en}$, $\tau : E \to \delta$ is a map from edges to actions of P s.t. for every edge $(u, v) \in E$ there exists an action $(\nu(u), \tau(u, v), \nu(v)) \in \delta$, and $\psi : V \to B$ is a map from nodes V to Boolean formulas over Var. A node v s.t. $\nu(v) = \mathsf{err}$ is called an *error node*.

A node $v \in V$ is *covered* iff there exists a node $u \in V$ that dominates v and there exists a set of nodes $X \subseteq V$, s.t. $\psi(u) \Rightarrow \bigvee_{x \in X} \psi(x)$ and $\forall x \in X \cdot \nu(u) = \nu(x) \wedge u \not\preceq x$, where $\preceq$ is the ancestor relation on nodes and all $x \in X$ are less than u according to some fixed total order on nodes V. A node u *dominates* v iff all paths from v_{en} to v pass through u. By convention, every node dominates itself.

Definition 1 (Well-labeledness of ARGs). *Given an ARG $\mathcal{A} = (V, E, v_{en}, \nu, \tau, \psi)$ of a program $P = (\mathcal{CP}, \delta, en, err, \mathsf{Var})$ and a map $\mathcal{L}$ from every $v \in V$ to a Boolean formula over Var, we say that $\mathcal{L}$ is a* well-labeling *of $\mathcal{A}$ iff (1) $\mathcal{L}(v_{en}) \equiv$ true; and (2) $\forall (u, v) \in E \cdot \mathcal{L}(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \mathcal{L}(v)'$. If ψ is a well-labeling of $\mathcal{A}$, we say that $\mathcal{A}$ is* well-labeled.

An ARG is *safe* iff for all $v \in V$ s.t. $\nu(v) = \mathsf{err}$, $\psi(v) \equiv$ *false*. An ARG is *complete* iff for all uncovered nodes u, for all $(\nu(u), T, \ell) \in \delta$, there exists an edge $(u, v) \in E$ s.t. $\nu(v) = \ell$ and $\tau(u, v) = T$.

Theorem 1 (Program Safety [1]). *If there exists a safe, complete, and well-labelled ARG for a program P, then P is safe.*

Abstract Domain. Abstract and concrete domains are often presented as Galois-connected lattices. In this paper, we use a more operational presentation. Without loss of generality, we restrict the concrete domain to a set B

```
 1: func VintaMain (Program P) :
 2:     create nodes v_en, v_err
 3:     ψ(v_en) ← true ; ν(v_en) ← en
 4:     ψ(v_err) ← false ; ν(v_err) ← err
 5:     marked(v_en) ← true
 6:     labels ← ∅
 7:     while true do
 8:         ExpandArg()
 9:         if ψ(v_err) is UNSAT then
10:             return SAFE
11:         labels ← Refine(A)
12:         if labels = ∅ then
13:             return UNSAFE

14: func GetFutureNode (ℓ ∈ CP) :
15:     if FN(ℓ) is defined then
16:         return FN(ℓ)
17:     create node v
18:     ψ(v) ← true ; ν(v) ← ℓ
19:     FN(l) ← v
20:     return v

21: func ExpandNode (v ∈ V) :
22:     if v has children then
23:         for all (v, w) ∈ E do
24:             FN(ν(w)) ← w
25:     else
26:         for all (ν(v), T, ℓ) ∈ δ do
27:             w ← GetFutureNode(ℓ)
28:             E ← E ∪ {(v, w)} ; τ(v, w) ← T

29: func ExpandArg () :
30:     vis ← ∅ ; FN ← ∅
31:     FN(err) ← v_err ; v ← v_en
32:     while true do
33:         ℓ ← ν(v)
34:         ExpandNode(v)
35:         if marked(v) then
36:             marked(v) ← false
37:             ψ(v) ← ComputePost(v)
38:             ψ(v) ← WidenWith({ψ(u) | u ∈ vis(ℓ)}, ψ(v))
39:             for all (v, w) ∈ E do marked(w) ← true
40:         else if labels(v) is defined then
41:             ψ(v) ← labels(v)
42:             for all {(v, w) ∈ E | labels(w) is undefined} do
43:                 marked(w) ← true
44:         vis(ℓ) ← vis(ℓ) ∪ {v}
45:         if v = v_err then break
46:         if Smt.IsValid(ψ(v) ⇒ ⋁_{u∈vis(ℓ),u≠v} ψ(u)) then
47:             erase FN(ℓ)
48:             repeat ℓ ← WtoExit(ℓ) until FN(ℓ) is defined
49:             v ← FN(ℓ) ; erase FN(ℓ)
50:             for all {(v, w) ∈ E | ∄u ≠ v · (u, w) ∈ E} do
51:                 erase FN(ν(w))
52:         else
53:             ℓ ← WtoNext(ℓ)
54:             v ← FN(ℓ) ; erase FN(ℓ)
```

Fig. 3. VINTA algorithm.

of all Boolean expressions over program variables (as opposed to the powerset of concrete program states). We define an abstract domain as a tuple $\mathcal{D} = (D, \top, \bot, \sqcup, \triangledown, \alpha, \gamma)$, where D is the set of abstract elements with two designated elements $\top, \bot \in D$, called *top* and *bottom*, respectively; two binary functions $\sqcup, \triangledown : D \times D \to D$, called *join* and *widen*, respectively; and two functions: an *abstraction* $\alpha : B \to D$ and a *concretization* $\gamma : D \to B$. The functions respect the expected properties: $\alpha(\mathit{true}) = \top$, $\gamma(\bot) = \mathit{false}$, for $x, y, z \in D$· if $z = x \sqcup y$ then $\gamma(x) \vee \gamma(y) \Rightarrow \gamma(z)$, etc. Note that D has no meet and no abstract order – we do not use them. Finally, we assume that for every action T, there is a sound abstract transformer $\textsc{Post}_{\mathcal{D}}$ s.t. if $d_2 = \textsc{Post}_{\mathcal{D}}(T, d_1)$ then $\gamma(d_1) \wedge [\![T]\!] \Rightarrow \gamma(d_2)'$, where $d_1, d_2 \in D$, and for a formula X, X' is X with all variables primed.

4 Vinta

In this section, we formally describe VINTA and discuss its properties.

4.1 Main Algorithm

VintaMain. Function VINTAMAIN in Fig. 3 implements the loop in Fig. 1. It takes a program $P = (\mathcal{CP}, \delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ and checks whether the error loca-

tion err is reachable.Without loss of generality, we assume that every location in $\mathcal{CP}$ is reachable from en and can reach err (ignoring the semantics of actions). VintaMain maintains a globally accessible ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$. If VintaMain returns SAFE, then $\mathcal{A}$ is safe, complete, and well-labeled (thus proving safety of P by Theorem 1).

VintaMain is parameterized by (1) the abstract domain $\mathcal{D}$, and (2) the refinement function Refine. First, an ARG is constructed by ExpandArg using an abstract transformer $\textsc{Post}_{\mathcal{D}}$. For simplicity of presentation, we assume that all labels are Boolean expressions that are implicitly converted to and from $\mathcal{D}$ using functions α and γ, respectively. ExpandArg always returns a complete and well-labeled ARG. So, on line 9, VintaMain only needs to check whether the current ARG is safe. If the check fails, Refine is called to find a counterexample and remove false alarms. We describe our implementation of Refine in Sec. 4.3, but the correctness of the algorithm depends only on the following abstract specification:

Definition 2 (Specification of Refine [1]). Refine *returns an empty map (*$\text{labels} = \emptyset$*) if there exists a feasible execution from* v_{en} *to* v_{err} *in* $\mathcal{A}$*. Otherwise, it returns a map* labels *from nodes to Boolean expressions s.t. (1)* $\text{labels}(v_{en}) \equiv \text{true}$ *and* $\text{labels}(v_{err}) \equiv \text{false}$*, and (2)* $\forall (u, v) \in E \cdot \text{labels}(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \text{labels}(v)'$.

In our case, refinement uses BMC and interpolation through an SMT solver to compute labels, therefore, if no labels are found, refinement produces a counterexample as a side-effect.

Whenever Refine returns a non-empty labeling (i.e., false alarms were removed), VintaMain calls ExpandArg again. ExpandArg uses *labels* to relabel the existing ARG nodes and uses $\textsc{Post}_{\mathcal{D}}$ to expand the ARG further, as necessary.

ExpandArg. ExpandArg constructs the ARG in a *recursive iteration strategy* [7], It assumes existence of a *weak topological ordering* (WTO) [7] of the CPG and two functions, WtoNext and WtoExit as described in Sec. 3.

ExpandArg maintains two local maps: vis and FN. vis maps a cutpoint ℓ to the set of visited nodes corresponding to ℓ, and FN maps a cutpoint ℓ to the first unexplored node $v \in V$ s.t. $\nu(v) = \ell$. The predicate *marked* specifies whether a node is labeled using AI (*marked* is *true*) or it gets a label from the map *labels* produced by Refine (*marked* is *false*). Marks are propagated from a node to children (lines 39 and 42). Initially, the entry node is marked (line 5), which causes all of its descendants to be marked as well. AI over all incoming edges of a node v is done using $\textsc{ComputePost}(v)$ that over-approximates $\textsc{Post}_{\mathcal{D}}$ computations over all predecessors of a node v (that are in vis).

Note that Vinta uses an ARG as an efficient representation of a disjunctive invariant: for each cutpoint $\ell \in \mathcal{CP}$, the disjunction $\bigvee_{v \in \mathsf{vis}(\ell)} \psi(v)$ is an inductive invariant. The key to efficiency is two-fold. First, a possibly expensive abstract subsumption check is replaced by an SMT-check (line 46). Second, inspired by [10], an expensive powerset widening is replaced by a simple widening scheme, WidenWith, that lifts base domain widening $\triangledown$ to a widening between a set and a *single* abstract element. We describe WidenWith in detail in Sec. 4.2.

VINTA is based on UFO [1], but improves it in two directions: (1) it extends UFO to arbitrary abstract domains using widening and (2) it employs a more efficient covering strategy (line 46). While in theory VINTA is compatible with the refinement strategy of UFO, in Sec. 4.3 we describe the shortcomings of UFO's refinement in our setting and present a new refinement strategy.

4.2 Widening

In this section, we describe the powerset widening operator WIDENWITH used by VINTA.

Definition 3 (Specification of WIDENWITH). *Let $\mathcal{D} = (D, \top, \bot, \sqcup, \triangledown, \alpha, \gamma)$ be an abstract domain. An operator $\triangledown_W : \mathcal{P}_f(D) \times D \to D$ is a* WIDENWITH *operator iff it satisfies the following two conditions:*

1. *(soundness) for any $X \subseteq D$ and $y \in D$, $(\gamma(X) \vee \gamma(y)) \Rightarrow (\gamma(X) \vee \gamma(X \triangledown_W y))$;*
2. *(termination) for any $X \subseteq D$, and a sequence $\{y_i\}_i \in D$, the sequence $\{Z_i\}_i \subseteq D$, where $Z_0 = X$, and $Z_i = Z_{i-1} \cup \{Z_{i-1} \triangledown_W y_i\}$ converges, i.e., $\exists i \cdot \gamma(Z_i) \Rightarrow \gamma(Z_{i+1})$,*

where $\gamma(X) \equiv \bigvee_{x \in X} \gamma(x)$, for some set of abstract elements X.

Note that unlike traditional powerset widening operators (e.g., [3]), WIDENWITH is defined for a pair of a set and an element (and not a pair of sets). It is inspired by the widening operator $\triangledown_T^p$ of Gulavani et al. [10], but differs from it in three important aspects. First, we do not require that if $z = \text{WIDENWITH}(X, y)$, then z is "bigger" than y, i.e., $\gamma(y) \Rightarrow \gamma(z)$. Intuitively, if X and y approximate sets of reachable states, then z over-approximates the *frontier* of y (i.e., states in y but not in X). Second, our termination condition is based on concrete implication (and not on an abstract order). Third, we do not require that X or the sets $\{Z_i\}_i$ in Def. 3 contain only "maximal" elements [10]. These differences give us more freedom in designing the operator and significantly simplify the implementation.

We now describe two implementations of WIDENWITH: the first, $\text{WIDENWITH}_\sqcup$, is based on $\triangledown_T^p$ from [10] and applies to any abstract domain while the second, $\text{WIDENWITH}_\vee$, requires an abstract domain that supports disjunction ($\vee$) and set difference ($\setminus$). One example of such a domain is BOXES [12]. The operators are defined as follows:

$$\text{WIDENWITH}_\sqcup(\emptyset, y) = y \qquad \text{WIDENWITH}_\vee(\emptyset, y) = y \tag{1}$$

$$\text{WIDENWITH}_\sqcup(X, y) = x \triangledown (x \sqcup y) \tag{2}$$

$$\text{WIDENWITH}_\vee(X, y) = \left((\bigvee X) \triangledown (\bigvee X \vee y)\right) \setminus \bigvee X \tag{3}$$

where $x \in X$ is picked non-deterministically from X.

Theorem 2 ($\text{WIDENWITH}_{\{\vee,\sqcup\}}$ Correctness). $\text{WIDENWITH}_\sqcup$ *and* $\text{WIDENWITH}_\vee$ *satisfy the two conditions of Def. 3.*

1: **func** UfoRef (ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$) :
2: $\quad \mathcal{L}_E \leftarrow$ EncodeBmc($\mathcal{A}$); $\mathcal{I} \leftarrow$ DagItp($(V, E, v_{\mathsf{en}}, v_{\mathsf{err}}), \mathcal{L}_E$); **return** DecodeBmc($\mathcal{I}$)

Fig. 4. Ufo refinement procedure.

4.3 Refinement

In this section, we formalize our refinement strategy. We start by reviewing the strategy used by Ufo and based on a concept of a *Restricted DAG Interpolant* (RDI) – an extension of a *path interpolant* [13, 15] to DAGs. In the rest of this section, we write F for a set of formulas; $G = (V, E, v^{en}, v^{ex})$ for a DAG with an entry node $v^{en} \in V$ and an exit node $v^{ex} \in V$, where v^{en} has no predecessors, v^{ex} has no successors, and every node $v \in V$ lies on a (v^{en}, v^{ex})-path. We also write $desc(v)$ and $anc(v)$ for the sets of descendants and ancestors of a node $v \in V$, respectively; $\mathcal{L}_E : E \to F$ and $\mathcal{L}_V : V \to F$ for maps from edges and vertices to formulas, respectively; and $FV(\varphi)$ for the set of free variables in a given formula φ.

Definition 4 (Restricted DAG Interpolant (RDI)). *Let G, $\mathcal{L}_E$, and $\mathcal{L}_V$ be as defined above. An RDI is a map $\mathcal{I} : V \to F$ s.t.*

1. $\forall e = (v_i, v_j) \in E \cdot \big(\mathcal{I}(v_i) \wedge \mathcal{L}_V(v_i) \wedge \mathcal{L}_E(e)\big) \implies \mathcal{I}(v_j) \wedge \mathcal{L}_V(v_j)$,
2. $\mathcal{I}(v^{en}) \equiv \text{true}$, *and* $\big(\mathcal{I}(v^{ex}) \wedge \mathcal{L}_V(v^{ex})\big) \equiv \text{false}$, *and*
3. $\forall v_i \in V \cdot FV(\mathcal{I}(v_i)) \subseteq \Big(\bigcup_{u \in desc(v_i)} FV(\mathcal{I}(u))\Big) \cap \Big(\bigcup_{u \in anc(v_i)} FV(\mathcal{I}(u))\Big)$.

Whenever $\forall v \cdot \mathcal{L}_V(v) = \mathit{true}$, we say that an RDI is *unrestricted* or simply a DAG Interpolant (DI). Intuitively, a DI $\mathcal{I}$ is a labeling of G such that for every path $v^{en}, \ldots, v^{ex}$, the sequence $\mathcal{I}(v^{en}), \ldots, \mathcal{I}(v^{ex})$ is a path interpolant [13, 15]. In general, in a proper RDI $\mathcal{I}$ (i.e., when $\exists v \cdot \mathcal{L}_V(v) \neq \mathit{true}$), $\mathcal{I}(v)$ is not an interpolant by itself, but is a projection of an interpolant to $\mathcal{L}_V(v)$. That is, $\mathcal{I}(v)$ is the restriction needed to turn $\mathcal{L}_V(v)$ into an interpolant. Thus, an RDI can be weaker (and possibly easier to compute) than a DI.

Ufo **Refinement.** Ufo's refinement procedure is shown in Fig. 4. It uses the procedure DagItp from [1][2] to compute a DI. Given an ARG $\mathcal{A} = (V, E, v_{\mathsf{en}}, \nu, \tau, \psi)$ with an error node v_{err}, it first constructs an edge labeling $\mathcal{L}_E$ using a BMC-encoding such that for each ARG edge e, $\mathcal{L}_E(e)$ is the semantics of the corresponding action $\tau(e)$ (i.e., $[\![\tau(e)]\!]$), with variables renamed and added as necessary, and such that for any path $v_1, \ldots, v_k$, the formula $\bigwedge_{i \in [1,k)} \mathcal{L}_E(v_i, v_{i+1})$ encodes all executions from v_1 to v_k. Many BMC-encodings can be used for this step, and we use the approach of [11]. For example, for the three edges (v_1, v_2^a),

[2] [1] used a different terminology. DagItp refers to the procedure in Thm. 3 of [1].

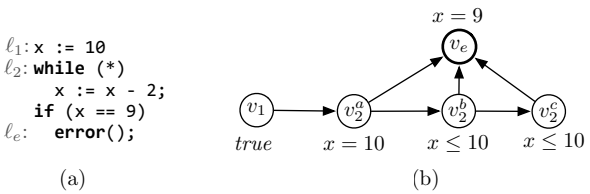

Fig. 5. (a) A program and (b) its ARG.

(v_2^a, v_e), (v_2^a, v_2^b) of the ARG in Fig. 2(c), the $\mathcal{L}_E$ map is

$$\mathcal{L}_E(v_1, v_2^a) \equiv x_0 = 0 \wedge y_0 = 0 \tag{4}$$

$$\mathcal{L}_E(v_2^a, v_e) \equiv x_\phi \geq 4 \wedge y_\phi \leq 2 \wedge x_\phi = x_0 \wedge y_\phi = y_0 \tag{5}$$

$$\begin{aligned}\mathcal{L}_E(v_2^a, v_2^b) \equiv\ & (x_1 = x_0 + 1 \wedge y_1 = y_0 + 1) \vee \\ & (x_0 \geq 4 \wedge x_1 = x_0 + 1 \wedge y_1 = y_0 + 1) \vee \\ & (x_1 = x_0 \wedge y_1 = y_0)\end{aligned} \tag{6}$$

where, in addition to renaming, two extra variables x_ϕ and y_ϕ were added for the SSA encoding since node v_e has multiple edges incident on it. $\mathcal{L}_E(v_1, v_2^a) \wedge \mathcal{L}_E(v_2^a, v_e)$ encodes all executions on the path v_1, v_2^a, v_e, and $\mathcal{L}_E(v_1, v_2^a) \wedge \mathcal{L}_E(v_2^a, v_2^b)$ encodes all executions on the path v_1, v_2^a, v_2^b. Second, the refined labels are computed as a DI $\mathcal{I} = \textsc{DagItp}((V, E, v_{\mathsf{en}}, v_{\mathsf{err}}), \mathcal{L}_E)$. Note that after reversing the renaming done by BMC-encoding (i.e., removing the subscripts), the DI $\mathcal{I}$ is a safe (by condition 2 of Def. 4) well-labeling (by condition 1 of Def. 4) of the ARG $\mathcal{A}$. Furthermore, $\mathcal{I}(v)$ is expressed completely in terms of variables defined before and used after $v \in V$. The result of refinement on our running example is shown in Fig. 2(d).

Using Ufo Refinement with Vinta. While Vinta can use Ufo's refinement since it satisfies the specification of Refine in Def. 2, we found that it does not scale in practice. We believe there are two key reasons for this.

The first reason is that the DI-based refinement uses just the ARG while completely ignoring its node labeling (i.e., the set of reachable states discovered by AI). Thus, while the DI-based refinement recovers from imprecision to remove false alarms, it may introduce imprecision for further exploration steps. For example, consider the program in Fig. 5(a) and its ARG in Fig. 5(b) produced by AI using the Box domain. The ARG has a false alarm (in reality, v_e is unreachable). A possible DI-based refinement changes the labels of v_2^b, v_2^c, and v_e to $x \leq 10 \wedge x \neq 9$, $x \neq 9$, and *false*, respectively. While this is sufficient to eliminate the false alarm, the new labels do not form an inductive invariant – thus further unrolling of the ARG is required. Note that the refinement "improved" the label of v_2^c to $x \neq 9$, but "lost" an important fact $x \leq 10$. Instead, we propose to restrict refinement to produce new labels that are stronger than the existing ones. In this example, such a restricted refinement would change the labels of v_2^b, v_2^c, and v_e to $x \leq 10 \wedge x \neq 9$, $x \leq 10 \wedge x \neq 9$, and *false*, thus completing the verification.

```
1: func VintaRef (ARG A = (V, E, v_en, ν, τ, ψ)) :
2:     L_E ← EncodeBmc(A) ; L_V ← Encode(ψ)
3:     I ← VintaRdi((V, E, v_en, v_err), L_E, L_V)
4:     if I = ∅ then return I
5:     for all v ∈ V do I(v) ← I(v) ∧ L_V(v)
6:     return DecodeBmc(I)
```

```
Require: L_V is a well-labeling of G
7: func VintaRdi (G, L_E, L_V) :
8:     for all e = (u, v) ∈ E do
9:         L_E(e) ← L_V(u) ∧ L_E(e)
10:    I ← DagItp(G, L_E)
11:    return I
```

Fig. 6. VintaRef refinement procedure.

The second reason is that ARGs produced by AI are large, and generating interpolants directly from them takes too long. Here, again, part of the problem is that refinement does not use the existing labeling to simplify the constraints. Instead of computing a DI of the ARG, we propose to compute an RDI restricted by the current labeling. Since an RDI is simpler (i.e., weaker, has fewer connectives, etc.) than a corresponding DI, the hope is that it is also easier to compute.

Vinta **Refinement.** Vinta's refinement procedure VintaRef is shown in Fig. 6. It takes a labeled ARG $\mathcal{A}$ and returns a new safe well-labeling *labels* of $\mathcal{A}$. First, it encodes the edges of $\mathcal{A}$ using BMC-encoding as described above (line 2). Second, the current labeling ψ of $\mathcal{A}$ is encoded to match the renaming introduced by the BMC-encoding. For example, for v_2^a in our running example, $\psi(v_2^a) \equiv x = 0 \wedge y = 0$, and the encoding $\mathcal{L}_V(v_2^a) \equiv x_0 = 0 \wedge y_0 = 0$. Third, it uses VintaRdi (shown in Fig. 6) to compute an RDI of $\mathcal{A}$ restricted by $\mathcal{L}_V$. Fourth, it turns the RDI into a DI by conjoining it with $\mathcal{L}_V$ (line 5). Finally, it decodes the labels by undoing the BMC-encoding (line 6).

The function VintaRdi computes an RDI by reducing it to computing a DI using the DagItp procedure from [1] described earlier. Note that it requires that $\mathcal{L}_V$ is a well-labeling, i.e., for all $(u, v) \in E$, $\mathcal{L}_V(u) \wedge \mathcal{L}_E(u, v) \Rightarrow \mathcal{L}_V(v)$. The idea is to "communicate" to the SMT-solver the restriction of node u by conjoining $\mathcal{L}_V(u)$ to every edge from u. This information might be helpful to the SMT-solver for simplifying its proofs and the resulting interpolants.

Theorem 3 (Correctness of VintaRef**).** VintaRef *satisfies the specification of* Refine *in Def. 2.*

There is a simple generalization of VintaRef: ψ on line 2 can be replaced by any over-approximation U of reachable states. The current invariant represented by the ARG is a good candidate and so are invariants computed by other techniques. The only restriction is that VintaRdi requires U to be a well-labeling. Removing this restriction from VintaRdi remains an open problem.

5 Implementation and Evaluation

5.1 Implementation

We have implemented Vinta in the Ufo framework [2] for verifying C programs, which is built on top of the LLVM compiler infrastructure [14]. Our modular implementation of Vinta allows abstract domains to be easily plugged in and

experimented with. Currently, the abstract domains used by VINTA are BOX and BOXES, defined in [12]. For SMT-solving and interpolation, VINTA uses Z3 [16] and MATHSAT5[3], respectively. In the rest of this section, we highlight the technical challenges addressed by our implementation. Specifically, we discuss our implementation of abstraction functions from Boolean expressions to BOX and BOXES elements, and describe key SMT-solving techniques that are instrumental to VINTA's efficiency. Our implementation and complete experimental results are available at `http://www.cs.toronto.edu/~aws/vinta`.

Abstraction Functions. We are using a simple abstraction function to convert between Boolean expressions and BOXES and BOX abstract domains. Given a formula φ, we first convert it to NNF. Then, we replace all literals involving more than one variable (e.g., $x + y = 0$) with *true*, thus over-approximating φ and removing all terms not expressible in BOX. Finally, for BOX, we additionally use join to approximate disjunction. This naive approach is very imprecise in general, but works well on our benchmarks.

Incremental Solving for Covering. Recall that EXPANDARG in Fig. 3 uses an SMT call at every cover check (line 46 in Fig. 3). This is highly inefficient. In practice, we exploit Z3's incremental interface (using `push` and `pop` commands) as follows. For each cutpoint ℓ, we maintain a separate SMT context $\texttt{ctx}_\ell$. Every time a node v s.t. $\nu(v) = \ell$ is not covered (i.e., the check on line 46 in Fig. 3 fails), $\neg\psi(v)$ is added to $\texttt{ctx}_\ell$. To check whether a node u with $\nu(u) = \ell$ is covered, we check whether $\psi(u)$ is satisfiable in $\texttt{ctx}_\ell$. If the result is UNSAT, then u is covered; otherwise, it is not covered and $\neg\psi(u)$ is added to $\texttt{ctx}_\ell$. Effectively, this is the same as checking whether $\psi(u) \wedge \bigwedge_{v \in \mathsf{vis}(\nu(u)), v \neq u} \neg\psi(v)$ is UNSAT, which is equivalent to line 46 of EXPANDARG.

Using Post Computations for Simplification. In our implementation, we keep track of those ARG edges for which $\textsc{Post}_{\mathcal{D}}$ computations returned $\bot$. For each such edge e, we can replace $\mathcal{L}_E(e)$ in VINTARDI with *false*, thus reducing the size of the formula.

Improving Interpolation with UNSAT Cores. One technical challenge we faced is that MATHSAT5's performance degrades significantly when interpolation support is turned on, particularly on large formulas. To reduce the size of the formula given to MATHSAT5, we use the *assumptions* feature in the highly efficient but lacking interpolation support Z3. Let a formula $\varphi_1 \wedge \ldots \wedge \varphi_n$ and a set $X = \{b_i\}_{i=1}^n$ of Boolean *assumptions* variables be given. When Z3 is passed a formula $\Phi = (b_1 \Rightarrow \varphi_1) \wedge \ldots \wedge (b_n \Rightarrow \varphi_n)$, it returns a subset of X, called UNSAT core, that has to be *true* to make Φ UNSAT. In our case, we add an assumption for each literal appearing in formulas in $\mathcal{L}_E$, and use Z3 to find unnecessary literals, i.e., those not in the UNSAT core. Since Z3 does not produce a minimal core, we repeat the minimization process three times. Finally, we set unnecessary literals to *true* and use MATHSAT5 to interpolate over the simplified formula.

[3] `http://mathsat.fbk.eu`

Algorithm	#Solved	#Safe	#Unsafe	Total Time (s)
vBox	**71**	20	**51**	**580** (539/**41**)
uBox	68	19	49	1,240 (1,162/78)
vBoxes	67	25	42	1,782 (596/1,186)
uBoxes	60	18	42	2,731 (808/1,923)
CpaAbe	65	**29**	36	1,167 (707/460)
CpaMemo	64	24	40	1,794 (**454**/1,341)
uInterp	70	20	50	1,535 (1,457/78)
uCp	69	19	50	1,687 (1,509/178)
uBp	64	15	49	1,062 (57/1,006)

Table 1. Summary of results on 93 C programs. Numbers in bold indicate the best result.

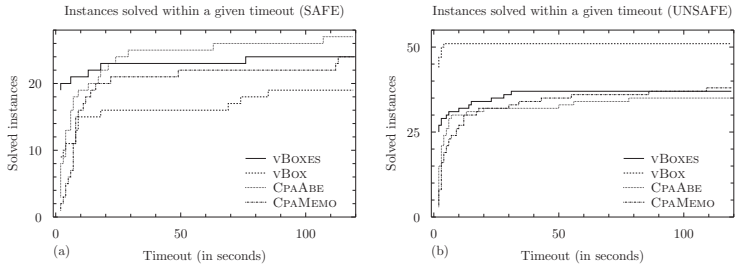

Fig. 7. Number of solved instances vs. timeout: (a) safe benchmarks; (b) unsafe benchmarks.

5.2 Evaluation

For evaluation, we used `ntdrivers-simplified`, `ssh-simplified`, and `systemc` benchmarks from the 2012 Software Verification Competition (SV-COMP 2012) [4]. In total, we had 93 C programs (41 safe and 52 buggy).

We implemented several instantiations of Vinta: vBox, vBoxes, uBox, and uBoxes, using the Box and Boxes domains, and VintaRef and UfoRef refinements, respectively. For Box and Boxes, we used the widening operators $\text{WidenWith}_{\sqcup}$ and $\text{WidenWith}_{\vee}$ from Sec. 4.2, respectively. In all cases, we applied widening on every third unrolling of each loop. We compared Vinta against the top two tools from SV-COMP 2012: CpaChecker-Abe (CpaAbe) and CpaChecker-Memo (CpaMemo), which are two variations of the predicate-abstraction-based software model checker CpaChecker [5]. For both tools, we used the same version and configuration as in the competition. We also compared against several instantiations of our Ufo framework: uInterp, uCp, and uBp, using interpolation-based verification by itself and in combination with Cartesian and Boolean predicate abstractions, respectively.

The overall results are summarized in Table 1. All experiments were conducted on a 3.40GHz Intel Core i7 processor with 8GB of RAM running Ubuntu Linux v11.10. We imposed a time limit of 500 seconds and a memory limit of 4GB per program. For each tool, we show the number of safe and unsafe instances solved and the total time taken. For example, vBox solved 20 safe and 51 unsafe examples in 580 seconds, spending 539s on safe ones and 41s on unsafe ones (time spent in unsolved instances is not counted). vBox is an overall winner, and is able to solve the most unsafe instances in the least amount of time. CpaAbe is

Program	vBoxes	uBoxes	vBox	uBox	CpaAbe	CpaMemo
s3_clnt_1	**0.30**	**0.30**	8.61	13.67	7.34	11.63
s3_clnt_2	**0.3**	**0.30**	8.79	13.45	6.72	8.53
s3_clnt_3	0.30	**0.29**	9.01	6.80	9.72	7.10
s3_clnt_4	**0.30**	**0.30**	9.55	8.52	6.33	12.43
s3_srvr_1a	**0.15**	–	1.08	–	2.86	4.344
s3_srvr_1b	**0.02**	**0.02**	–	–	1.49	1.64
s3_srvr_1	**0.00**	**0.00**	**0.00**	**0.00**	21.21	8.63
s3_srvr_2	**0.64**	115.48	–	115.13	63.44	113.07
s3_srvr_3	**0.75**	123.57	69.70	123.61	17.23	22.55
s3_srvr_4	**0.59**	168.44	85.81	168.08	7.50	14.57
s3_srvr_6	473.15	319.00	**74.87**	359.39	181.82	–
s3_srvr_7	**13.82**	–	–	274.12	24.84	112.53
s3_srvr_8	**0.69**	78.53	245.52	76.12	18.48	8.82
token_ring.01	**0.94**	–	4.05	–	4.13	8.04
token_ring.02	**2.53**	–	18.29	–	6.69	49.11
token_ring.03	**6.06**	–	–	–	29.55	–
token_ring.04	**18.22**	–	–	–	146.43	–
token_ring.05	**76.29**	–	–	–	–	–
token_ring.06	–	–	–	–	–	–
token_ring.07	–	–	–	–	–	–
token_ring.08	–	–	–	–	–	–

Table 2. Time of running Vinta, CpaAbe, and CpaMemo on 21 safe benchmarks. '–' indicates a timeout.

the winner on the safe instances, with vBoxes coming in second. In the rest of this section, we examine these results in more detail.

Instances Solved vs. Timeout. Fig. 7 shows the number of instances solved in a given timeout for (a) safe and (b) unsafe benchmarks, respectively. To avoid clutter, we omit uInterp, uBp, and uCp from the graphs and restrict the timeout to 120s, since only a few instances took more time. For the safe cases, vBoxes is a clear winner for the timeout of $\leq$ 10s. Indeed, on most safe benchmarks, vBoxes takes a lot less time to complete than CpaAbe, CpaMemo, and all other instantiations of Ufo and Vinta. For the unsafe cases, vBox is a clear winner for all timeouts. Interestingly, the extra precision of Boxes makes vBoxes perform poorly on unsafe instances: it either solves an unsafe instance in one iteration (i.e., no refinement), or runs out of time in the first AI- or refinement-phase.

Detailed Comparison. We now examine a portion of the benchmark suite in more detail, specifically, safe `ssh-simplified` benchmarks and safe `token_ring` benchmarks (from `systemc`). Table 2 shows the time taken by the different instantiations of Vinta, CpaAbe, and CpaMemo. On these benchmarks, we observe that vBoxes outperforms all other approaches.

Compared with CpaAbe and CpaMemo, vBoxes is able to solve almost all instances in much less time. For example, on `token_ring.05`, both CpaAbe and CpaMemo fail to return a result, but vBoxes proves safety in 76 seconds. Similarly, vBoxes is superior on most `ssh-simplified` examples.

To understand the importance of the refinement strategy, consider the `ssh-simplified` benchmarks. The invariant for most `ssh-simplified` instances is computable using Boxes with an appropriate widening strategy ("widen on every fourth unrolling"). The results in the table show how Vinta's refinement strategy is able to recover precision when an inadequate refinement strategy is

used (i.e., "widen on every third unrolling"). Using UFO's refinement, uBOXES takes substantially more time and more iterations or fails to return a result within the allotted time limit. For example, on `s3_srvr_2`, vBOXES requires a single refinement, whereas uBOXES requires 38. Positive effects of VINTA's AI-guided refinement are also visible in vBOX vs. uBOX.

In summary, our results demonstrate the power of VINTA's refinement strategy and show how basic instantiations of VINTA can compete and outperform highly-optimized verification tools like CPACHECKER. To further improve VINTA's performance, it would be interesting to experiment with other abstract domains as well as with different automatic strategies for choosing an appropriate domain. For example, we saw that BOXES, in comparison with BOX, generates very large ARGs for unsafe examples. One strategy would be to keep track of ARG size and time spent in refinement and revert to a less precise abstract domain like BOX when they become too large.

6 Conclusion

In this paper, we presented VINTA, an iterative algorithm that uses Craig interpolants to refine invariants produced by abstract interpretation and eliminate false alarms. VINTA's verification technique marries the efficiency of abstract interpretation with the precision of bounded model checking and the ability to "guess" invariants of interpolation-based verification.

Our evaluation of VINTA against state-of-the-art verification tools demonstrates the power of our approach and calls for further experimentation with our refinement strategy on different abstract domains.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-approximations to Over-approximations and Back. In: Proc. of TACAS'12. LNCS, vol. 7214, pp. 157–173 (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: UFO: A Framework for Abstraction- and Interpolation-Based Software Verification. In: Proc. of CAV'12 (2012), to appear
3. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening Operators for Powerset Domains. STTT 8(4-5), 449–466 (2006)
4. Beyer, D.: Competition On Software Verification - (SV-COMP). In: Proc. of TACAS'12. LNCS, vol. 7214, pp. 504–524 (2012), `http://sv-comp.sosy-lab.org/`
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Proc. of CAV'11. LNCS, vol. 6806, pp. 184–190 (2011)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: TACAS'99. LNCS, vol. 1579. Springer (1999)
7. Bourdoncle, F.A.: Efficient Chaotic Iteration Strategies with Widenings. In: Proc. of FMPA'93. pp. 128–141. LNCS (1993)
8. Craig, W.: Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. J. of Symbolic Logic 22(3), 269–285 (1957)

9. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: A New Algorithm for Property Checking. In: Proc. of FSE'06. pp. 117–127 (2006)
10. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 443–458 (2008)
11. Gurfinkel, A., Chaki, S., Sapra, S.: Efficient Predicate Abstraction of Program Summaries. In: Proc. of NFM'11. LNCS, vol. 6617, pp. 131–145 (2011)
12. Gurfinkel, A., Chaki, S.: Boxes: A Symbolic Abstract Domain of Boxes. In: Proc. of SAS'10. LNCS, vol. 6337, pp. 287–303 (2010)
13. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proc. of POPL'04. pp. 232–244 (2004)
14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. of CGO'04. pp. 75–88 (2004)
15. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136 (2006)
16. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)
17. Wang, C., Yang, Z., Gupta, A., Ivancic, F.: Using Counterexamples for Improving the Precision of Reachability Computation with Polyhedra. In: Proc. of CAV'07. LNCS, vol. 4590, pp. 352–365 (2007)