From Under-approximations to Over-approximations and Back

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. Current approaches to software model checking can be divided into over-approximation-driven (OD) and under-approximation-driven (UD). OD approaches maintain an abstraction of the transition relation of a program and use abstract reachability to build an inductive invariant (or find a counterexample). At the other extreme, UD approaches attempt to construct inductive invariants by generalizing from finite paths through the control-flow graph of the program.

In this paper, we present Ufo, an algorithm that unifies OD and UD approaches in order to leverage both of their advantages. Ufo is parameterized by the degree to which over- and under-approximations drive the analysis. At one extreme, Ufo is a novel interpolation-based (UD) algorithm that generates interpolants to label (refine) multiple program paths using a single SMT solver query. At the other extreme, Ufo uses an abstract domain to drive the analysis, while using interpolants to strengthen the abstraction.

We have implemented Ufo in LLVM and applied it to programs from the Competition on Software Verification. Our experimental results demonstrate the utility of our algorithm and the benefits of combining UD and OD approaches.

1 Introduction

In recent years, we have witnessed a divergence in software model checking techniques. Traditionally, as promoted by the SLAM project [3], software model checkers implemented a variant of the counterexample-guided abstraction refinement (CEGAR) [11] loop, where over-approximating abstractions of programs are computed. In cases where spurious counterexamples are introduced by over-approximation, refinement is used to eliminate them. We henceforth categorize such techniques as *over-approximation-driven* (OD). OD techniques mainly rely on *predicate abstraction* [16] for computing an abstract post operator. In the refinement stage, new predicates (facts) are added to build more precise abstractions. OD techniques can supply us with efficient safety proofs, when relatively coarse abstractions are sufficient to prove correctness. Unfortunately, it is often the case that a large number of predicates is required to reach a deep error or to compute an inductive invariant, causing the abstraction step to be very expensive.

On the other hand, *under-approximation-driven* (UD) software model checking techniques are becoming more popular. Such techniques attempt to construct a program invariant by generalizing from finite program paths. For example, in [24], McMillan uses *Craig Interpolants*, derived from proofs of unsatisfiability of paths to an error location, to compute program invariants. In our previous work [2], we used predicate abstraction to generalize symbolic program executions. Note that SYNERGY [17] and DASH [4] are considered under-approximation-driven according to our categorization, as they use weakest-precondition computations along infeasible symbolic paths to refine a partition of the state space with the goal of computing an invariant. Testing in these techniques only acts as a way of choosing which symbolic paths to examine. UD techniques avoid the expensive step of computing an abstract post operator, giving them an advantage over OD techniques. Unfortunately, due to the fact that they are not driven by an abstract domain, they may have to examine a large number of program paths to compute an inductive invariant or find an erroneous execution.

In this paper, our goal is to resolve the disconnect between OD and UD approaches to software model checking. Specifically, we present UFO, a software model checking algorithm for sequential programs that is parameterized by the degree to which over- and under-approximations drive the analysis. UFO makes two contributions: (1) it combines UD and OD approaches, and (2) at the UD extreme, it is a novel interpolation-based algorithm that generates interpolants to label (refine) multiple program paths using a single SMT solver query. This allows UFO to exploit an SMT solver's ability to enumerate program executions, giving it an advantage over other interpolation-based algorithms, e.g., [24, 23], that explicitly enumerate program paths.

We have implemented UFO in the LLVM compiler infrastructure [22] and experimented with various instantiations of it on benchmarks from the Competition on Software Verification [5]. Our experimental results show the utility of our interpolation-based algorithm. Moreover, they show that augmenting UFO with an abstract domain (e.g., predicate abstraction) often outperforms both the OD and UD extremes.

The rest of the paper is organized as follows: In Sec. 2, we illustrate the operation of UFO on an example. In Sec. 3, we provide the definitions and notation used in the paper. In Sec. 4, we present the UFO algorithm. In Sec. 5, we present the refinement procedure. In Sec. 6, we describe our UFO implementation and present our experimental evaluation. In Sec. 7, we place UFO in the context of related work. Finally, in Sec. 8, we conclude the paper and outline directions for future work.

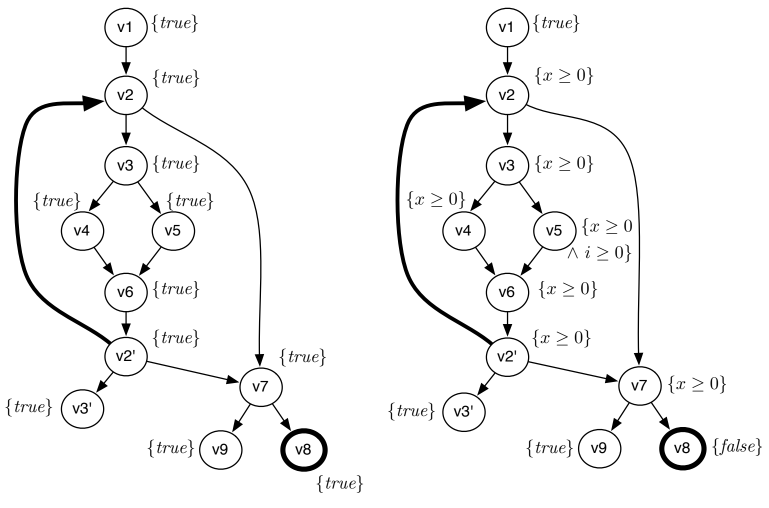
2 Overview

The core of UFO is a UD algorithm parameterized by an abstract POST operator and a novel interpolation-based refinement procedure. In this section, we illustrate the novel parts of UFO by instantiating POST to always return *true*, the weakest admissible POST. In practice, we also instantiate POST with Boolean and Cartesian predicate abstractions.

Consider function `foo` shown in Fig. 1(a), which takes `n` as a parameter. We want to prove that location 8 with label `ERROR` is unreachable for any value of `n`.

```
foo(int n):
1 i = 0, x = 0;
2 while (i < n)
3    if (i <= 2)
4       x = 0;
     else
5       x = i;
6    i = i + 1;
7 if (x < 0)
8    ERROR:
9 return;
```

(a)



(b) (c)

Fig. 1. (a) Safe function `foo`. ARGs for `foo` (b) 1st iteration; (c) after refinement.

Constructing the ARG. UFO starts by constructing an *Abstract Reachability Graph* (ARG) for the program. The ARG for `foo` is shown in Fig. 1(b). Each node in the ARG relates to a location in `foo`. For example, nodes v_2 and v'_2 represent location 2. UFO associates a label $\{\varphi_i\}$ for each node v_i, where φ_i is an over-approximation of the set of reachable states at v_i.

UFO expands the ARG using the *recursive iteration strategy* [8]. That is, the innermost loop is unrolled first. In our example, UFO starts by creating nodes v_1 and v_2. Following the recursive iteration strategy, it enters the loop, creating nodes v_3, v_4, v_5, v_6, and v'_2. All nodes are initially labelled with *true*, the weakest possible over-approximation.

Upon reaching node v'_2, UFO adds v'_3 and v_7 as children of v'_2. At this point, the label of v'_2 is subsumed by the one of v_2. We say that v'_2 is *covered* by v_2 and show it as the bold back-edge in Fig. 1(b). Therefore, UFO exits the loop and goes on to process node v_7, adding v_8 (`ERROR` node) and v_9.

Refining the ARG. Once the ARG has been completely expanded, UFO checks if the label on the `ERROR` node v_8 is UNSAT. The label is *true*, which means that there is a potential execution to location 8 that either does not enter the loop at v_2 or takes one iteration before exiting through v'_2. To check if such an execution exists, UFO constructs a formula representing all executions in the ARG: for each node v_i that can reach v_8, it creates the following formula μ_i:

$$\mu_1 : c_{v_1} \Rightarrow (i_0 = 0 \wedge x_0 = 0 \wedge c_{v_2})$$
$$\mu_2 : c_{v_2} \Rightarrow ((i_0 < n \wedge c_{v_3}) \vee (i_0 \geq n \wedge x_4 = x_0 \wedge c_{v_7}))$$
$$\mu_3 : c_{v_3} \Rightarrow ((i_0 \leq 2 \wedge c_{v_4}) \vee (i_0 > 2 \wedge c_{v_5}))$$
$$\mu_4 : c_{v_4} \Rightarrow (x_1 = 0 \wedge x_3 = x_1 \wedge c_{v_6})$$
$$\mu_5 : c_{v_5} \Rightarrow (x_2 = i_0 \wedge x_3 = x_2 \wedge c_{v_6})$$
$$\mu_6 : c_{v_6} \Rightarrow (i_1 = i_0 + 1 \wedge c_{v'_2})$$
$$\mu'_2 : c_{v'_2} \Rightarrow (i_1 \geq n \wedge x_4 = x_3 \wedge c_{v_7})$$
$$\mu_7 : c_{v_7} \Rightarrow (x_4 \geq 0 \wedge c_{v_8})$$

For example, μ_2 specifies that if control reaches node v_2 (represented by the Boolean *control variable* c_{v_2}), then either $i_0 < n$ and control goes to v_3, or $i_0 \geq n$ and control goes to v_7. To avoid naming conflicts, each time a variable appears on the left hand side of an assignment, it is given a fresh subscript (e.g., x becomes x_1 in μ_4).

The formula $c_{v_1} \wedge \mu_1 \wedge \cdots \wedge \mu_7$ is UNSAT. Hence, there is no feasible execution in the ARG that can reach v_8. At this point, *Craig interpolants* [13] are used to relabel the ARG. Given a pair of formulas (A, B) s.t. $A \wedge B$ is UNSAT, an interpolant for (A, B) is a formula I s.t. $A \Rightarrow I$, $I \Rightarrow \neg B$, and I is over the variables shared between A and B. To derive a new label for v_7, UFO sets $A = c_{v_1} \wedge \mu_1 \wedge \cdots \wedge \mu_2'$ and $B = \mu_7$. A possible interpolant I for (A, B) is $c_{v_7} \wedge x_4 \geq 0$. To remove the instrumentation variable c_{v_7}, UFO sets it to *true*. It also removes the subscript from x_4 to arrive at a formula $x \geq 0$ over program variables. The new labels generated by interpolants are shown in Fig. 1(c). In Sec. 5, we formalize the process of deriving labels from interpolants and prove that for any edge (v_i, v_j) in the ARG, the resulting labels for v_i and v_j form a Hoare triple with respect to the program statement on the edge.

Note that in Fig. 1(c), v_8 is labelled with $\{\mathit{false}\}$ and v_2' is still covered, since its label $x \geq 0$ is subsumed by the label on v_2. Therefore, UFO terminates execution declaring `foo` safe. When applied to this example, the algorithm in [24] requires at least two refinements, as the control-flow graph (CFG) is unrolled into a tree, thus creating two paths to `ERROR` through each branch of the conditional statement (location 3). UFO, on the other hand, unrolls the CFG into a DAG and exploits the power of SMT solvers for enumerating paths.

3 Abstract Reachability Graphs

Here, we present the notation and definitions used in the rest of the paper.

Programs. A *program* P is a tuple $(\mathcal{L}, \Delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$, where $\mathcal{L}$ is a finite set of control locations, Δ is a finite set of actions, $\mathsf{en} \in \mathcal{L}$ is the entry location of P, $\mathsf{err} \in \mathcal{L}$ is the error location, and Var is the set of variables of program P. An *action* $(\ell_1, T, \ell_2) \in \Delta$ represents an edge in the control flow graph of P, where $\ell_1, \ell_2 \in \mathcal{L}$ and T is a program statement. We assume that there does not exist an action $(\ell_1, T, \ell_2) \in \Delta$ s.t. $\ell_1 = \mathsf{err}$.

A program statement is either an assume statement `assume`(Q), where Q is a Boolean expression over Var, or an assignment statement `x = `E, where `x` is a variable in Var and E is an expression over the variables in Var. We use the notation $[\![T]\!]$ to denote the standard semantics of a program statement T. For example, for an assignment statement `x = x + 1`, $[\![\texttt{x = x + 1}]\!]$ is $x' = x + 1 \wedge \forall y \in \mathsf{Var} \cdot y \neq x \Rightarrow y' = y$. For a formula ϕ, we use ϕ' to denote ϕ with all variables replaced by their primed versions.

We say that a program P is *safe* iff there does not exist a feasible execution that starts in en and reaches err through the actions in Δ.

Weak Topological Ordering. A *Weak Topological Ordering* (WTO) [8] 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 *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_s be a component with head v. Suppose that U_s is a subcomponent of some component U. If there exists a $u \in U$ s.t. $u \notin U_s$ 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. Now suppose that U_s 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. $u \notin U_s$ 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 function `foo` in Fig. 1(d), a WTO of the control locations is 1 (2 3 4 5 6) 7 8 9, where 2 is the head of the component comprising the while loop. $\textsc{WtoNext}(2) = 3, \textsc{WtoNext}(6) = 2$, and $\textsc{WtoExit}(2) = 7$. Note that $\textsc{WtoNext}$ and $\textsc{WtoExit}$ are partial functions and we only use them where they have been defined.

Abstract Reachability Graphs (ARGs). Let $P = (\mathcal{L}, \Delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ be a program. A *Reachability Graph* (RG) of P is a tuple $(V, E, v_{\mathsf{en}}, \nu, \tau)$, where (V, E, v_{en}) represents a directed acyclic graph (DAG) rooted at the *entry node* $v_{\mathsf{en}} \in V$, $\nu : V \to \mathcal{L}$ is a map from nodes to control locations of P, where $\nu(v_{\mathsf{en}}) = \mathsf{en}$, and $\tau : E \to \Delta$ is a map from edges to actions of P s.t. for every edge $(u, v) \in E$, there exists an action $(\nu(u), \tau(u, v), \nu(v)) \in \Delta$.

An *Abstract Reachability Graph* (ARG) $\mathcal{A}$ of P is a tuple $(U, \psi, \sqsubseteq, \sqsubseteq_t)$, where $U = (V, E, v_{\mathsf{en}}, \nu, \tau)$ is an RG of P, ψ is a map from nodes V to formulas over Var, $\sqsubseteq$ is the ancestor relation over the nodes of U, and $\sqsubseteq_t$ is a fixed linearization of the topological ordering of the nodes of U. 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 $X \subset V$, where $\psi(u) \Rightarrow \bigvee_{x \in X} \psi(x)$ and $\forall x \in X \cdot x \sqsubset u \wedge \nu(x) = \nu(u)$. A node u *dominates* v iff all paths from v_{en} to v pass through u. Every node v dominates itself.

Definition 1 (Well-labeledness of ARGs). *An ARG $\mathcal{A} = (U, \psi, \sqsubseteq, \sqsubseteq_t)$, where $U = (V, E, v_{en}, \nu, \tau)$, for a program $P = (\mathcal{L}, \Delta, en, err, \mathsf{Var})$ is* well-labelled *iff (1) $\psi(v_{en}) \equiv$ true; and (2) $\forall (u, v) \in E, \psi(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \psi(v)'$.*

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

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

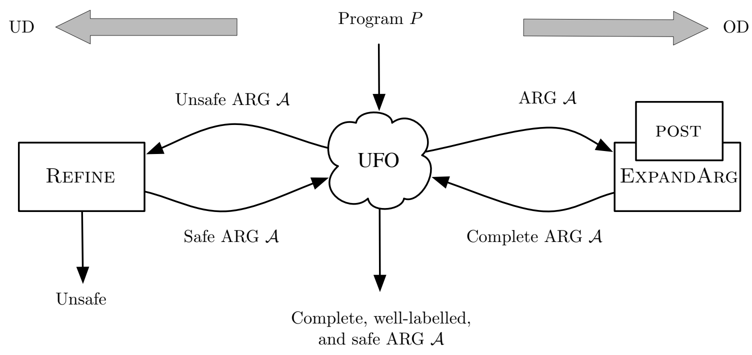


Fig. 2. High level description of UFO.

```
func UfoMain (Program $P$) :
    create node $v_{\mathsf{en}}$
    $\psi(v_{\mathsf{en}}) \leftarrow \mathit{true}, \nu(v_{\mathsf{en}}) \leftarrow \mathsf{en}$
    $\mathit{marked}(v_{\mathsf{en}}) \leftarrow \mathit{true}$
    $\mathit{labels} \leftarrow \emptyset$
    while $\mathit{true}$ do
        ExpandArg()
        if $\psi(v_{\mathsf{err}})$ is UNSAT then
            return SAFE
        $\mathit{labels} \leftarrow$ Refine()
        if $\mathit{labels} = \emptyset$ then
            return UNSAFE
        clear $\mathsf{AH}$ and $\mathsf{FN}$

func GetFutureNode ($\ell \in \mathcal{L}$) :
    if $\mathsf{FN}(\ell)$ exists then
        return $\mathsf{FN}(\ell)$
    create node $v$
    $\psi(v) \leftarrow \mathit{true}; \nu(v) \leftarrow \ell$
    $\mathsf{FN}(l) \leftarrow v$
    return $v$

func ExpandNode ($v \in V$) :
    if $v$ has children then
        for all $(v, w) \in E$ do
            $\mathsf{FN}(\nu(w)) \leftarrow w$
    else
        for all $(\nu(v), T, \ell) \in \Delta$ do
            $w \leftarrow$ GetFutureNode($\ell$)
            $E \leftarrow E \cup \{(v, w)\}; \tau(v, w) \leftarrow T$

func ExpandArg () :
    $v \leftarrow v_{\mathsf{en}}$
    while $\mathit{true}$ do
        ExpandNode($v$)
        if $\mathit{marked}(v)$ then
            $\mathit{marked}(v) \leftarrow \mathit{false}$
            $\psi(v) \leftarrow \bigvee_{(u,v) \in E}$ Post$(u, v)$
            for all $(v, w) \in E$ do $\mathit{marked}(w) \leftarrow \mathit{true}$
        else if $\mathit{labels}(v)$ bound then
            $\psi(v) \leftarrow \mathit{labels}(v)$
            for all $\{(v, w) \in E \mid \mathit{labels}(w)$ unbound$\}$ do
                $\mathit{marked}(w) \leftarrow \mathit{true}$
        if $v = v_{\mathsf{err}}$ then break
        if $\nu(v)$ is head of a component then
            if $\psi(v) \Rightarrow \bigvee_{u \in \mathsf{AH}(\nu(v))} \psi(u)$ then
                erase $\mathsf{AH}(\nu(v))$ and $\mathsf{FN}(\nu(v))$
                $l \leftarrow$ WtoExit($\nu(v)$)
                $v \leftarrow \mathsf{FN}(l)$; erase $\mathsf{FN}(l)$
                for all $\{(v, w) \in E \mid \nexists u \neq v \cdot (u, w) \in E\}$ do
                    erase $\mathsf{FN}(\nu(w))$
                continue
            add $v$ to $\mathsf{AH}(\nu(v))$
        $l \leftarrow$ WtoNext($\nu(v)$)
        $v \leftarrow \mathsf{FN}(l)$; erase $\mathsf{FN}(l)$
```

Fig. 3. The UFO Algorithm. Implementation of REFINE is presented in Sec. 5.

The proof of this theorem follows from Theorem 1 in [24].

4 The UFO Algorithm

In this section, we describe our verification algorithm UFO that takes a program P with a designated error location v_{err} and determines whether v_{err} is reachable. The output of the algorithm is either an execution of P that ends in v_{err}, or a complete, well-labeled, and safe ARG of P. The novelty of UFO lies in its combination of UD and OD techniques. Fig. 2 illustrates the two main states of UFO: (1) exploring (OD), and (2) generalizing (UD). Exploring an ARG is done by unwinding the CFG while *computing* node labels using an abstract post operator POST. Generalizing is done by *guessing* (typically using interpolants) a safe labelling of the current ARG from a proof of infeasibility of unsafe executions in $\mathcal{A}$.

The pseudo-code for the algorithm is given in Fig. 3. Function EXPANDARG (line 29) is responsible for the exploration, and REFINE (line 10) is used for generalization. Note that UFO is parameterized by POST (line 35) – more precise POST makes it more OD-like and less precise POST – more UD-like. We present the main parts of the algorithm in this section, and an implementation of REFINE in Sec. 5.

Main Loop. UFOMAIN is the main function of UFO. It receives a program $P = (\mathcal{L}, \Delta, \mathsf{en}, \mathsf{err}, \mathsf{Var})$ as input and attempts to prove that P is safe (or unsafe) by constructing a complete, well-labelled, and safe ARG for P (or by finding an execution to err). The function EXPANDARG is used to construct an ARG $\mathcal{A} = (U, \psi, \sqsubseteq, \sqsubseteq_T)$ for P. By definition, it always constructs a complete, well-labelled ARG. Line 8 of UFOMAIN checks if the result of EXPANDARG is a safe ARG by checking whether the label on the node v_{err} is satisfiable (by construction, v_{err} is the only node in $\mathcal{A}$ s.t. $\nu(v_{\mathsf{err}}) = \mathsf{err}$). If $\psi(v_{\mathsf{err}})$ is UNSAT, then $\mathcal{A}$ is safe, and UFO terminates by declaring the program safe (following Theorem 1). Otherwise, REFINE is used to compute new labels. In Definition 2, we provide a specification of REFINE that maintains the soundness of UFO. In Sec. 5, we present a refinement algorithm satisfying Definition 2.

Definition 2 (Specification of Refine). *If there exists a feasible execution to v_{err} in $\mathcal{A}$, then* REFINE *returns an empty map (*$\text{labels} = \emptyset$*). Otherwise, it returns a map from nodes to labels s.t.* $\text{labels}(v_{err}) \equiv \text{false}$*,* $\text{labels}(v_{en}) \equiv \text{true}$*, and* $\forall (u, v) \in E' \cdot \text{labels}(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \text{labels}(v)'$*, where E' is E restricted to edges along paths to v_{err}. That is, the labeling precludes erroneous executions and maintains well-labelledness of $\mathcal{A}$ (per Definition 1).*

Constructing the ARG. EXPANDARG adopts a standard *recursive iteration strategy* [8] for unrolling a CFG into an ARG. To do so, it makes use of a *weak topological ordering* (WTO) [8] of program locations. A recursive iteration strategy starts by unrolling the innermost loops until "stabilization", i.e., until a loop head is covered, before exiting to the outermost loops. We assume that the first location in the WTO is en and the last one is err.

EXPANDARG maintains two global maps: AH (active heads) and FN (future nodes). For a loop head l, $\mathsf{AH}(l)$ is the set of nodes $V_\ell \subseteq V$ for location l that are heads of the component being unrolled. When a loop head is covered (line 43), all active heads belonging to its location are removed from AH (line 44). FN maps a location to a single node and is used as a worklist, i.e., it maintains the next node to be explored for a given location. Example 1 demonstrates the operation of EXPANDARG.

Example 1. Consider the process of constructing the ARG in Fig. 1(b) for function `foo` in Fig. 1(a). When ExpandArg processes node v_2' (i.e., when $v = v_2'$ at line 31), $\mathsf{AH}(2) = \{v_2\}$, since the component (2 3 4 5 6) representing the loop is being unrolled and v_2 is the only node for location 2 that has been processed. When Ufo covers v_2' (line 43), it sets $\mathsf{AH}(2) = \emptyset$ (line 44) since the component has stabilized and Ufo has to exit it. Here, WtoExit(2) = 7, so Ufo continues processing from node $v_7 = \mathsf{FN}(7)$ (the node for the first location after the loop).

Suppose Refine returned a new label for node v. When ExpandArg updates $\psi(v)$ (line 38), it *marks* all of its children that do not have labels in *labels*. This is used to strengthen the labels of v's children w.r.t the refined over-approximation of reachable states at v, using the operator Post (line 35). ExpandArg only attempts to cover nodes that are loop heads. It does so by checking if the label on a node v is subsumed by the labels on $\mathsf{AH}(\nu(v))$ (line 43). If v is covered, Ufo exits the loop (line 45); otherwise, it adds v to $\mathsf{AH}(\nu(v))$.

Post Operator. Ufo is parameterized by the abstract operator Post. For sound implementations of Ufo, Post should take an edge (u, v) as input and return a formula ϕ s.t. $\psi(u) \wedge [\![\tau(u, v)]\!] \Rightarrow \phi'$, thus maintaining well-labelledness of the ARG. In the UD case, Post always returns *true*, the weakest possible abstraction. In the combined UD+OD case, Post is driven by an abstract domain, e.g., based on predicate abstraction.

Theorem 2 (Soundness). *Given a program P, if* Ufo *run on P terminates with SAFE, the resulting ARG $\mathcal{A}$ is safe, complete, and well-labelled. If* Ufo *terminates with UNSAFE, then there exists an execution that reaches* err *in P.*

5 Refinement

In this section, we present our refinement procedure Refine. It begins by computing a formula φ, called an *ARG condition*, representing all executions in a given ARG. If φ is unsatisfiable, Refine invokes an interpolation-based algorithm is to compute new labels for the ARG.

ARG Condition. Given an ARG $\mathcal{A}$ with an entry and an error nodes v_{en}, v_{err}, respectively, we define ArgCond as follows:

$$\textsc{ArgCond}(v_{\mathsf{err}}) \triangleq c_{u_1} \wedge \mu_1 \wedge \cdots \wedge \mu_n, \tag{1}$$

$$\text{where } \mu_i = (c_{u_i} \Rightarrow \bigvee_{(u_i, w) \in E} (c_w \wedge \mathit{encode}(u_i, w))), \tag{2}$$

$u_1 = v_{\mathsf{en}}$, and $u_1, \ldots, u_n$ is the sequence of all nodes, excluding v_{err}, that can reach v_{err} in $\mathcal{A}$, ordered by $\sqsubseteq_t$, and c_{u_i} is a fresh Boolean *control variable* representing the node u_i.

If $\mathit{encode}(\cdot, \cdot)$ is *true*, then $\textsc{ArgCond}(v_{\mathsf{err}})$ is satisfiable iff there exists a path from v_{en} to v_{err} in $\mathcal{A}$. $\mathit{encode}(u, v)$ is a formula describing the semantics of an edge (u, v) that is used to restrict satisfying assignments of $\textsc{ArgCond}(v_{\mathsf{err}})$ to feasible executions. For example, for function `foo` in Fig. 1(a), we encode the statement on edge (v_4, v_6) as follows: $\textit{encode}(v_4, v_6) = (x_1 = 0 \wedge x_3 = x_1)$. where x_1 is a fresh name for variable x, and $x_3 = x_1$ is used to equate the name of x at node v_6 (which is x_3) with the value of x after executing this edge.

For the purpose of presentation, we provide a simplified definition of *encode*. In practice, we use the SSA-based encoding defined in [18]. Let $\mathsf{SVar} = \{x_v \mid x \in \mathsf{Var} \wedge v \in V\}$ be the set of variables that can appear in $\textit{encode}(\cdot, \cdot)$. That is, for each variable $x \in \mathsf{Var}$ and node $v \in V$, we create a *symbolic variable* $x_v \in \mathsf{SVar}$. The map $\mathsf{SMap} : \mathsf{SVar} \to \mathsf{Var}$ associates each x_v with its program variable x. The predicate $\mathsf{inScope} : \mathsf{SVar} \times V$ is defined so that $\mathsf{inScope}(x_u, v)$ holds iff $u = v$. If $\mathsf{inScope}(x_u, v)$ holds, we say that x_u is *in-scope* at node v; otherwise, it is *out-of-scope* at v.

Definition 3 (encode). *For an edge $(u, v) \in E$: If $\tau(u, v)$ is an assignment statement `x = E`, then* $\mathrm{encode}(u, v) = (x_v = E[x \leftarrow x_u]) \wedge \forall y \in \mathsf{Var} \cdot y \neq x \Rightarrow y_v = y_u$. *If $\tau(u, v)$ is an assume statement `assume`(Q), then* $\mathrm{encode}(u, v) = Q[x \leftarrow x_u \mid x \in \mathrm{var}(Q)] \wedge \forall y \in \mathsf{Var} \cdot y_v = y_u$, *where* $\mathrm{var}(Q)$ *is the set of variables appearing in Q.*

For example, for an edge $(u, v) \in E$ s.t. $\tau(u, v)$ is `x = x + 1`, $\textit{encode}(u, v) = x_v = x_u + 1 \wedge y_v = y_u$, assuming $\mathsf{Var} = \{x, y\}$.

Lemma 1. *Given an ARG $\mathcal{A}$, there exists a total onto map from satisfying assignments of* $\textsc{ArgCond}(v_{\mathsf{err}})$ *to feasible program executions from v_{en} to v_{err}.*

Labels from Interpolants. Given an ARG $\mathcal{A}$ with error node v_{err}, when $\textsc{ArgCond}(v_{\mathsf{err}})$ is unsatisfiable, $\textsc{Refine}$ must return a set of labels for the nodes of the ARG that satisfy well-labelledness conditions and the specification of $\textsc{Refine}$ (Definitions 1 and 2). We now show how to extract such labels from an interpolant sequence of $\textsc{ArgCond}(v_{\mathsf{err}})$. For a sequence of formulas $A_1, \ldots, A_n$ s.t. $\bigwedge_{i \in [1,n]} A_i$ is UNSAT, an *interpolant sequence* [24, 10] $I_1, \ldots, I_{n+1}$ is defined as follows: (1) $I_1 \equiv \textit{true}$, (2) $\forall i \in [1, n] \cdot I_i \wedge A_i \Rightarrow I_{i+1}$, (3) I_i is over the variables shared between $A_1, \ldots, A_{i-1}$ and $A_i, \ldots, A_n$, and (4) $I_{n+1} \equiv \textit{false}$.

Let $I_1, \ldots, I_{n+1}$ be an interpolant sequence for the sequence of formulas $(c_{u_1} \wedge \mu_1) \wedge \mu_2 \wedge \cdots \wedge \mu_n$ constituting $\textsc{ArgCond}(v_{\mathsf{err}})$. By definition, an interpolant I_i is an over-approximation of the set of states at nodes in $u_i, \ldots, u_n$ that are directly reachable from states at nodes in $u_1, \ldots, u_{i-1}$. For node u_i, this includes all states reachable at u_i since all incoming edges to u_i are from nodes that are topologically before it, i.e., $u_1, \ldots, u_{i-1}$.

Example 2. Consider node v_2' in Figure 1(c). An interpolant for v_2' is $I_2' = (c_{v_2'} \wedge x_3 \geq 0) \vee (c_{v_7} \wedge x_4 \geq 0)$. Informally, I_2' specifies that either execution reaches v_2' with $x_4 \geq 0$, or it reaches v_7 with with $x_4 \geq 0$. v_7 states appear in the formula because v_7 is directly reachable from node v_2 which comes before v_2' in the topological order. $\square$

For each node u_i, our goal is to extract the set of reachable states at u_i from the interpolant I_i. For instance, for node v_2' from Example 2, we want to extract

$x_3 \geq 0$, the set of reachable states at v'_2, from the interpolant I'_2. To do so, we use the following transformation:

$$\textsc{Clean}(I_i) \triangleq \\ \forall \{x \mid x \in \mathit{var}(I_i) \wedge \neg \mathit{inScope}(x, u_i)\} \cdot \forall \{c_{u_j} \mid u_j \in V\} \cdot I[c_{u_i} \leftarrow \top], \quad (3)$$

where $\mathit{var}(I_i)$ is the set of variables appearing in I_i.

Example 3. Continuing Example 2, $\textsc{Clean}(I'_2) = \forall x_4, c_{v_7} \cdot I[c_{v'_2} \leftarrow \top] = x_3 \geq 0$. x_4 is quantified out since it is out-of-scope at v'_2. By replacing each variable y in the resulting formula with $\mathsf{SMap}(y)$, we get the label $\{x \geq 0\}$ for v'_2, as shown in Figure 1(c).

By definition, $\textsc{Clean}(I_i)$ is a formula over the variables in-scope at u_i. Theorem 3 states that the labels produced by $\textsc{Clean}$ result in a safe ARG and satisfy well-labelledness properties. That is, for any two nodes u_i, u_j, where there is an edge (u_i, u_j), the labels produced for u_i and u_j form a Hoare triple w.r.t the statement $\tau(u_i, u_j)$ encoded as $\mathit{encode}(u_i, u_j)$.

Theorem 3. *Let* $I'_k = \textsc{Clean}(I_k)$*. (a) If* $k = 1$*, then* $I'_k \equiv \text{true}$*, and if* $k = n$*, then* $I'_k \equiv \text{false}$*. (b) For any two nodes* $u_i, u_j \in V$ *s.t.* $(u_i, u_j) \in E$*,* $I'_i \wedge \text{encode}(u_i, u_j) \Rightarrow I'_j$*, where* $I'_j = \textsc{Clean}(I_j)$*.*

Proof. Part (a) follows from the definition of an interpolant sequence.
Part (b):

$$\begin{aligned}
& I_i \wedge \mu_i \wedge \cdots \wedge \mu_{j-1} \Rightarrow I_j \\
& \quad \text{(set } c_{u_i} \text{ to } \top \text{ and logic)} \\
\Rightarrow\ & I_i[c_{u_i} \leftarrow \top] \wedge c_{u_j} \wedge \mathit{encode}(u_i, u_j) \wedge \mu_{i+1} \wedge \cdots \wedge \mu_{j-1} \Rightarrow I_j \\
& \quad \text{(let } \Pi = \{c_{u_{i+1}}, \ldots, c_{u_{j-1}}\}) \\
\Rightarrow\ & I_i[c_{u_i} \leftarrow \top, \Pi \leftarrow \bot] \wedge c_{u_j} \wedge \mathit{encode}(u_i, u_j) \Rightarrow I_j \\
& \quad \text{(set } c_{u_j} \text{ to } \top) \\
\Rightarrow\ & I_i[\Pi \leftarrow \bot, c_{u_i} \leftarrow \top, c_{u_j} \leftarrow \top] \wedge \mathit{encode}(u_i, u_j) \Rightarrow I_j[c_{u_j} \leftarrow \top] \\
& \quad \text{(use } (\forall x . f) \Rightarrow f) \\
\Rightarrow\ & I'_i \wedge \mathit{encode}(u_i, u_j) \Rightarrow I_j[c_{u_j} \leftarrow \top] \\
& \quad \text{(out-of-scope variables of } u_j \text{ are not in the antecedent)} \\
\Rightarrow\ & I'_i \wedge \mathit{encode}(u_i, u_j) \Rightarrow I'_j
\end{aligned}$$

□

Finally, $\textsc{Refine}$ returns the labeling map $\{u_i \mapsto \textsc{Clean}(I_i)' \mid i \in [1, n+1]\}$, where $u_{n+1} = v_{\mathsf{err}}$ and $\textsc{Clean}(I_i)' = \textsc{Clean}(I_i)[x \leftarrow \mathsf{SMap}(x) \mid x \in \mathsf{SVar}]$.

In summary, our refinement technique uses a *single* SMT query φ to decide feasibility of *all* unsafe executions of an ARG $\mathcal{A}$. When φ is unsatisfiable, it extracts a new labeling for $\mathcal{A}$ that rules out all infeasible unsafe executions from an interpolant sequence of φ .

6 Implementation and Evaluation

Implementation. We have implemented Ufo in the LLVM compiler infrastructure [22] and used it to verify properties of C programs from the 2012

Competition on Software Verification [5]. We used MathSat4 [9] for SMT-checking and interpolation, and Z3 [25] for quantifier elimination. Our implementation, benchmarks, and complete experimental results are available at `http://www.cs.toronto.edu/~aws/ufo`.

We used LLVM to heavily optimize all input programs prior to analysis. Because the benchmarks are meant for verification tools, these optimizations might be unsound with respect to the intended verification semantics. However, in all but one case (`pipeline`), our verification results are as expected: we find a bug in buggy programs, and prove safety of safe ones. Furthermore, we have implemented a proof and a counterexample checker that verify that the results produced by UFO are sound with respect to our semantics of C. All results discussed here have been validated by an appropriate checker.

Evaluation. For the evaluation, we used the `ntdrivers-simplified`, `ssh-simplified`, and `systemc` benchmarks from [5], and the pacemaker benchmarks from [1]. Overall, we had 105 C programs: 48 safe and 57 buggy. All experiments were conducted on an Intel Xeon 2.66GHz processor running a 64-bit Linux, with a 300 second time and 4GB memory limits per program, respectively.

We have evaluated 5 configurations of Ufo: (1) a pure UD, called uUfo, where Post always returns *true*; (2) with Cartesian predicate abstraction, called cpUfo; (3) with Boolean predicate abstraction, called bpUfo; (4) a pure OD with Cartesian predicate abstraction, called Cp, and a pure OD with Boolean predicate abstraction, called Bp. Note that Boolean predicate abstraction is more precise, but is exponentially more expensive than Cartesian abstraction.

The results are summarized in Table 1. For each configuration, we show the number of instances solved (#Solved), number of safe (#Safe) and unsafe (#Unsafe) instances solved, number of unsound results (#Unsound), where a result is unsound if it does not agree with the benchmark categorization in [5], and the total time.

On these benchmarks, cpUfo performs significantly better than all other configurations, both in total time and number of instances solved. The uUfo configuration is a close second. We have also compared our results against the UD tool Wolverine [21] that implements a version of Impact [24] algorithm. All configurations of Ufo perform significantly better than Wolverine.

Furthermore, we compared our tool against the results of the extensive study reported in [7] for the state-of-the-art OD tools CpaChecker [7], Blast [6], and SatAbs [12]. Both uUfo and cpUfo configurations are able to solve all buggy `transmitter` examples. However, according to [7], CpaChecker, Blast, and SatAbs are unable to solve most of these examples, even though they are run on a faster processor with a 900s time limit and 16GB of memory. Additionally, on the `ntdrivers-simplified`, uUfo, cpUfo and bpUfo perform significantly better than all of the aforementioned tools.

Table 2 presents a detailed comparison between different configurations of Ufo on 32 (out of 105) programs. In the table, we show time, number of iterations (#Iter), and time spent in interpolation (#iTime) and post (#pTime), respectively. Times taken by other parts of the algorithm (such as Clean) were

Algorithm	#Solved	#Safe	#Unsafe	#Unsound	Total Time (s)
uUfo	78	22	56	0	8,289
cpUfo	79	22	57	1	7,838
bpUfo	69	17	52	1	11,260
Cp	49	10	39	0	15,363
Bp	71	19	52	1	10,018
Wolverine	38	18	20	5	19,753

Table 1. Summary of results on 105 C programs.

insignificant and are omitted. Cp configuration was not able to solve all but one of these examples, and is omitted as well.

In this sample, cpUfo is best overall, however, it is often not the fastest approach on any given example. This is representative of its performance over the whole benchmark. As expected, both uUfo and cpUfo spend most of their time in computing interpolants, while bpUfo and Bp spend most of their time in predicate abstraction.

The results show that there is clearly a synergy between UD and OD-driven parts of the analysis. For example, in `toy1_BUG` and `s3_srvr_1a`, predicate abstraction decreases the number of required iterations. Several of the buggy examples from the `token_ring` family cannot be solved by a UD-only uUfo configuration alone. However, there are also some interactions. For many of the safe cases that require a few iterations, uUfo performs better than other combinations. For many unsafe cases that bpUfo can solve, it performs much better alone than in a combination.

In summary, our results show that the novel UD-driven algorithm that underlies Ufo (uUfo configuration) is very effective compared to the state-of-the-art approaches. Furthermore, there is a clear synergy in combining UD and OD approaches, with cpUfo performing the best overall. However, there are also some interactions where the combination does not result in the best of the individual approaches. Managing these interactions effectively is the subject of future work.

7 Related Work

In this section, we place Ufo in the context of related work. Specifically, we compare it with the most related UD and OD verification techniques.

Ufo is based on a novel interpolation-driven verification algorithm. It extends Impact [24], by unrolling the program into a DAG instead of a tree and by using a single SMT query to both discharge all infeasible unsafe executions and to compute new labels. In effect, Ufo uses the SMT solver to enumerate acyclic program paths, whereas Impact enumerates those paths explicitly. Furthermore, Ufo extends Impact by using an abstract post operator during exploration. As we show in our experiments, this can lead to fewer iterations and faster verification.

We have recently developed an inter-procedural extension of Impact, called Whale [1]. Whale works on loop-free recursive programs and uses interpolants derived from DAG encodings of a procedure to compute procedure summaries.

	uUfo			cpUfo				bpUfo				Bp			
Program	Time	Iter	iTime	Time	#Iter	iTime	pTime	Time	Iter	iTime	pTime	Time	Iter	iTime	pTime
Unsafe Programs															
kundu1	-	-	-	24.22	4	20.3	1.84	122.88	4	56.9	54.66	33.39	3	20.23	10.95
kundu2	1.24	2	1.16	2.74	2	2.08	0.6	8.15	2	1.2	5.66	8.6	2	3.49	4.3
s3_srvr_11	1.91	4	1.67	2.78	4	1.58	0.89	118.41	4	1.72	112.76	4.25	3	2.6	1.37
s3_srvr_12	4.17	4	3.85	5.07	3	3.44	1.36	5.36	3	3.58	1.44	8.19	3	5.85	1.91
token_ring.08	12.34	4	11.84	13.5	4	11.07	1.91	19.64	3	3.7	14.62	14.15	3	1.85	11.12
token_ring.09	12.54	4	11.98	22.66	4	19.72	2.35	-	-	-	-	167.49	3	3.85	157.58
token_ring.10	15.6	4	15.05	14.02	3	11.99	1.69	-	-	-	-	-	-	-	-
token_ring.11	29.69	4	29.08	22.47	4	18.52	3.19	156.76	3	4.57	145.99	66.59	3	4.21	58.68
token_ring.12	26.94	4	26.31	13.98	3	11.45	2.15	-	-	-	-	-	-	-	-
token_ring.13	36.56	4	35.76	34.17	4	29.38	4.02	-	-	-	-	-	-	-	-
token_ring.14	10.3	3	9.99	33.49	4	29.17	3.59	-	-	-	-	-	-	-	-
token_ring.15	51.79	4	51.11	34.19	4	29.18	4.17	-	-	-	-	-	-	-	-
toy1	96.49	10	89.08	79	9	68.04	6.98	13.54	3	4.77	7.96	-	-	-	-
toy2	12.83	5	12.24	60.73	8	50.71	6.14	-	-	-	-	-	-	-	-
ddd3	0.66	4	0.5	0.19	2	0.04	0.11	0.18	2	0.03	0.11	0.27	2	0.05	0.2
Safe Programs															
pc_sfifo_1	-	-	-	-	-	-	-	3.51	3	2.24	0.79	-	-	-	-
s3_clnt_1	11.03	10	8.18	15.68	10	8.2	4.5	-	-	-	-	14.52	5	1.92	8.21
s3_clnt_2	16	10	11.35	20.02	10	10.86	4.67	-	-	-	-	-	-	-	-
s3_clnt_3org	28.87	11	17.35	37.08	11	17.6	8.17	-	-	-	-	-	-	-	-
s3_clnt_3	13.02	10	9.14	17.42	10	9.01	4.45	-	-	-	-	-	-	-	-
s3_clnt_4	13.4	10	9.62	17.45	10	9.16	4.58	-	-	-	-	-	-	-	-
s3_srvr_1a	5.2	10	2.95	5.16	8	2.32	1.07	0.76	4	0.17	0.39	0.43	3	0.07	0.26
s3_srvr_1b	1.37	7	1	2.9	7	1.71	0.69	0.89	5	0.47	0.28	-	-	-	-
s3_srvr_2	171.15	17	116.82	184.01	17	112.65	18.65	-	-	-	-	-	-	-	-
s3_srvr_3	133.07	17	99.96	147.55	17	98.69	16.02	-	-	-	-	33.71	5	1.07	21.18
s3_srvr_4	-	-	-	-	-	-	-	-	-	-	-	8	4	0.74	5.36
s3_srvr_8	101.4	14	76.6	115.08	14	73.9	17.62	-	-	-	-	-	-	-	-
token_ring.01	98.18	18	81.58	23.64	10	17.72	1.78	0.69	4	0.27	0.23	0.69	4	0.19	0.31
token_ring.02	-	-	-	-	-	-	-	2.15	4	0.71	0.7	2.63	4	1.06	0.59
token_ring.03	-	-	-	-	-	-	-	76.18	4	4.74	37.66	-	-	-	-
token_ring.04	-	-	-	-	-	-	-	-	-	-	-	152.62	4	10.82	2.45
token_ring.05	-	-	-	-	-	-	-	-	-	-	-	149.35	4	8.25	97.48

Table 2. Results of running Ufo on 33 programs from the benchmarks. All times are in seconds.

The intra-procedural technique presented here is orthogonal to that of Whale, and the interpolation-based refinement presented here is new. It would be interesting to see if the combination of UD and OD in Ufo can be adapted to the inter-procedural setting of Whale.

Dash [4] uses weakest-precondition (WP) over infeasible program paths to partition (i.e., refine) a program's state space. In contrast, Ufo refines multiple program paths at the same time. Moreover, Ufo uses interpolants for refinement, an approach that has been shown to provide more relevant predicates than WP-based refinement [20]. We believe that our multi-path refinement strategy can be easily implemented in Dash to generate test-cases for multiple frontiers or split different regions at the same time.

At a high level, Ufo is similar to the abstract algorithm Smash [15], in the sense that it combines over- and under-approximations. In [15], the only instantiation of Smash that is experimented with is an under-approximation-driven algorithm based on Dash [4], where no abstract domain is used. In this paper, we have experimented with multiple instances of Ufo, ranging from UD to OD. Other differences between Smash and Ufo include the fact that Ufo refines multiple paths at the same time, whereas Smash considers a single path at a time.

Lazy abstraction [19] is the closest OD algorithm to Ufo. Ufo can be seen as extending lazy abstraction in two directions. First, Ufo unrolls a program into a DAG (and not a tree). Second, it uses all the labels produced by interpolation, and only applies predicate abstraction to the "frontier" nodes that are not known to reach an error location.

We are not the first to apply interpolation to multiple program paths. In [14], Esparza et al. use interpolants to find predicates that eliminate multiple spurious counterexamples simultaneously. Their algorithm uses an eager abstraction-refinement loop, and a BDD-based interpolation procedure. In contrast, the refinement in Ufo uses an SMT-solvers-based interpolation procedure. It is not clear whether BDD-based techniques of [14] can be efficiently adapted to the SMT-based setting.

8 Conclusion

Software model checkers can be divided into over-approximation-driven (OD) (e.g., Slam [3]) and under-approximation-driven (UD) (e.g., Impact [24]). An OD software model-checker maintains an abstraction of the transition relation of a program and uses abstract reachability to build an inductive invariant or find a counterexample. A UD model checker avoids the (potentially expensive) abstraction step, and instead attempts to guess an inductive invariant by generalizing from finite paths through the control-flow graph of the program. Until now, combinations of these techniques have not been explored.

In this paper, we presented Ufo – a model checking algorithm that tightly couples UD and OD approaches. At the core of Ufo is a UD algorithm that is parameterized by an abstract Post operator, and a novel interpolation-based refinement procedure. The refinement procedure uses a *single* SMT query to decide feasibility of *all* unsafe executions in an unrolling of a program's CFG.

We have implemented Ufo within LLVM [22], and experimented with two variants of Post based on Boolean and Cartesian predicate abstractions. We have evaluated our implementation on benchmarks from the Competition on Software Verification [5]. Our results show that Ufo is very competitive compared to the state-of-the-art. There is a clear synergy in combining UD and OD approaches. However, there are also undesirable interactions. We believe that this work opens new avenues for exploring combinations of UD- and OD-based approaches to verification, a direction we hope to explore in the future.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: "Whale: An Interpolation-based Algorithm for Inter-procedural Verification". In: Proc. of VMCAI'12 (to appear) (2012)
2. Albarghouthi, A., Gurfinkel, A., Wei, O., Chechik, M.: "Abstract Analysis of Symbolic Executions". In: Proc. of CAV'10. LNCS, vol. 6174, pp. 495–510 (2010)

3. Ball, T., Rajamani, S.: "The SLAM Toolkit". In: Proc. of CAV'01. LNCS, vol. 2102, pp. 260–264 (2001)
4. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J.: "Proofs from Tests". In: Proc. of ISSTA'08. pp. 3–14 (2008)
5. Beyer, D.: "Competition On Software Verification" (2012), `http://sv-comp.sosy-lab.org/`
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: "The Software Model Checker Blast". STTT 9(5-6), 505–525 (2007)
7. Beyer, D., Keremoglu, M.E.: "CPAchecker: A Tool for Configurable Software Verification". In: Proc. of CAV'11 (2011)
8. Bourdoncle, F.A.: "Efficient Chaotic Iteration Strategies with Widenings". In: Proc. of FMPA'93. pp. 128–141. LNCS (1993)
9. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: "The MathSAT 4 SMT Solver". In: Proc. of CAV'08. pp. 299–303 (2008)
10. Cimatti, A., Griggio, A., Sebastiani, R.: "Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories". ACM Trans. Comput. Log. 12(1), 7 (2010)
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: "Counterexample-Guided Abstraction Refinement". In: Proc. of CAV'00. LNCS, vol. 1855, pp. 154–169 (2000)
12. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: "SATABS: SAT-based Predicate Abstraction for ANSI-C". In: Proc. of TACAS'05. LNCS, vol. 3440, pp. 570–574 (2005)
13. 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)
14. Esparza, J., Kiefer, S., Schwoon, S.: "Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems". In: Proc. of TACAS'06. pp. 489–503 (2006)
15. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: "Compositional May-Must Program Analysis: Unleashing the Power of Alternation". In: Proc. of POPL'10. pp. 43–56 (2010)
16. Graf, S., Saïdi, H.: "Construction of Abstract State Graphs with PVS". In: Proc. of CAV'97. vol. 1254, pp. 72–83 (1997)
17. 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)
18. Gurfinkel, A., Chaki, S., Sapra, S.: "Efficient Predicate Abstraction of Program Summaries". In: Proc. of NFM'11. LNCS, vol. 6617, pp. 131–145 (2011)
19. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: "Lazy Abstraction". In: Proc. of POPL'02. pp. 58–70 (2002)
20. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: "Abstractions from Proofs". In: Proc. of POPL'04. pp. 232–244 (2004)
21. Kroening, D., Weissenbacher, G.: "Interpolation-Based Software Verification with Wolverine". In: Proc. of CAV'11. LNCS, vol. 6806, pp. 573–578 (2011)
22. Lattner, C., Adve, V.: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: CGO'04 (2004)
23. McMillan, K.: "Lazy Annotation for Program Testing and Verification". In: Proc. of CAV'10. LNCS, vol. 6174, pp. 104–118 (2010)
24. McMillan, K.L.: "Lazy Abstraction with Interpolants". In: Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136 (2006)
25. de Moura, L., Bjørner, N.: "Z3: An Efficient SMT Solver". In: Proc. of TACAS'08. LNCS, vol. 4963, pp. 337–340 (2008)