Whale: An Interpolation-based Algorithm for Inter-procedural Verification

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. In software verification, Craig interpolation has proven to be a powerful technique for computing and refining abstractions. In this paper, we propose an interpolation-based software verification algorithm for checking safety properties of (possibly recursive) sequential programs. Our algorithm, called Whale, produces inter-procedural proofs of safety by exploiting interpolation for guessing function summaries by generalizing under-approximations (i.e., finite traces) of functions. We implemented our algorithm in LLVM and applied it to verifying properties of low-level code written for the pacemaker challenge. We show that our prototype implementation outperforms existing state-of-the-art tools.

1 Introduction

In the software verification arena, software model checking has emerged as a powerful technique both for proving programs correct and for finding bugs. Given a program P and a safety property φ to be verified, e.g., an assertion in the code, a model checker either finds an execution of P that refutes φ or computes an invariant that proves that P is correct w.r.t. φ.

Traditionally [3], software model checkers rely on computing a finite abstraction of the program, e.g., a Boolean program, and using classical model checking algorithms [8] to explore the abstract state space. Due to the over-approximating nature of these abstractions, the found counterexamples may be spurious. Counterexample-guided abstraction refinement (CEGAR) techniques [7] help detect these and refine the abstraction to eliminate them. This loop continues until a real counterexample is found or a proof of correctness, in the form of a program invariant, is computed.

More recently, a new class of software model checking algorithms has emerged. They construct program invariants by generalizing from finite paths through the control flow graph of the program. The most prominent of these are *interpolation-based algorithms* [27, 26, 16], introduced by McMillan in [27] and inspired by the success of *Craig interpolants* [9] for image-approximation in symbolic model checking [25]. In general, interpolation-based software model checking techniques extract interpolants from refutation proofs of infeasible program paths. The interpolants form an inductive sequence of Hoare triples that prove safety of a given program path, and potentially others.

Interpolation-based techniques avoid the expensive abstraction step of their traditional CEGAR-based counterparts and, due to their reliance on examining program paths for deriving invariants, are better suited for bug finding [26]. Yet, so far, interpolation-based techniques have been limited to intra-procedural analysis [27], restricted to non-recursive programs with bounded loops [26], or not modular in terms of generated proofs [16].

In this paper, we present WHALE: an inter-procedural interpolation-based software model checking algorithm that produces modular safety proofs of (recursive) sequential programs. Our key insight is to use interpolation to compute a function summary by generalizing from an under-approximation of a function, thus avoiding the need to fully expand the function and resulting in modular proofs of correctness. The use of interpolants allows us to produce concise summaries that eliminate facts irrelevant to the property in question. We also show how the power of SMT solvers can be exploited in our setting by encoding a path condition over multiple (or all) inter-procedural paths of a program in a single formula. We have implemented a prototype of WHALE using the LLVM compiler infrastructure [23] and verified properties of low-level C code written for the pacemaker grand challenge.

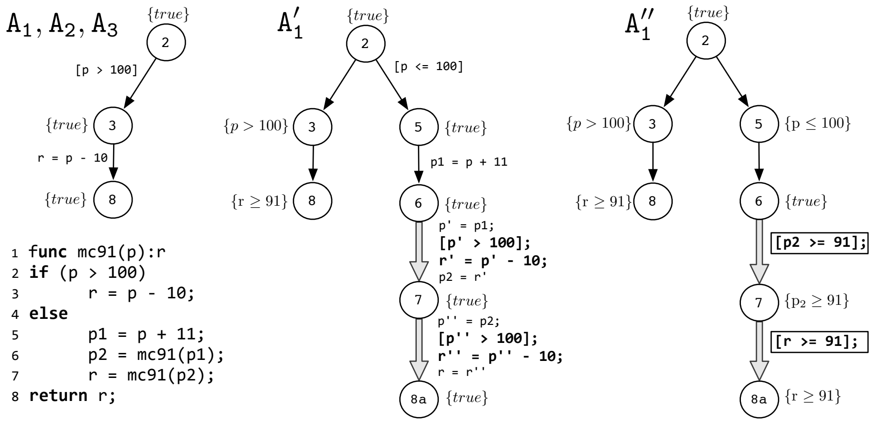
The rest of this paper is organized as follows: In Sec. 2, we illustrate WHALE on an example. In Sec. 3, we present background and notation used in the rest of the paper. In Sec. 4, we introduce inter-procedural reachability graphs. In Sec. 5, we present the algorithm. In Sec. 6, we discuss our implementation and present our experimental results. Finally, in Sec. 7 and Sec. 8, we discuss related work, sketch future research directions, and conclude the paper.

2 Motivating Example

In this section, we use WHALE to prove that `mc91` in Fig. 1, a variant of the famous McCarthy 91 function [24], always returns a value ≥ 91, i.e., $\texttt{mc91(p)} \geq 91$ for all values of p.

WHALE works by iteratively constructing a forest of *Abstract Reachability Graphs* (ARGs) (we call it an *iARG*) with one ARG for the main function, and one ARG for each function call inside each ARG. Each ARG $\mathcal{A}_i$ is associated with some function F_k, an expression G_i over the arguments of F_k, called the *guard*, and an expression S_i over the arguments and the return variables of F_k, called the *summary*. Intuitively, WHALE uses ARG $\mathcal{A}_i$ to show that function F_k behaves according to S_i, assuming the arguments satisfy G_i and assuming all other functions behave according to their corresponding ARGs in the iARG. A node v in an ARG $\mathcal{A}_i$ corresponds to a control location ℓ_v and is labeled by an expression e_v over program variables. WHALE maintains the invariant that e_v is an over-approximation of the states reachable from the states in G_i, at the entry point of F_k, along the path to v. It is always sound to let e_v be *true*. We now apply WHALE to `mc91` in Fig. 1, producing ARGs `A` (starting with $\texttt{A}_1$), with `G` and `S` as their guards and summaries, respectively.

Step 1. For each ARG in Fig. 1, the number inside a node v is the location ℓ_v and the expression in braces is e_v. For our property, $\texttt{mc91(p)} \geq 91$, the guard $\texttt{G}_1$ is *true*, and the summary $\texttt{S}_1$ is $r \geq 91$. The single path of $\texttt{A}_1$ is a potential



```
1 func mc91(p):r
2 if (p > 100)
3       r = p - 10;
4 else
5       p1 = p + 11;
6       p2 = mc91(p1);
7       r = mc91(p2);
8 return r;
```

Fig. 1. Applying WHALE to `mc91`.

counterexample: it reaches the return statement (line 8), and node 8 is labeled *true* (which does not imply the summary $r \geq 91$). To check for feasibility of the computed counterexample, WHALE checks satisfiability of the corresponding *path formula* $\pi = true \wedge (p > 100) \wedge (r = p - 10) \wedge (r < 91)$ obtained by conjoining the guard, all of the conditions and assignments on the path, and the negation of the summary. Here, π is unsatisfiable. Hence, the counterexample is infeasible, and the ARG labeling can be strengthened to exclude it.

Step 2. Like [27], WHALE uses interpolants to strengthen the labels. For a pair of formulas (A, B) s.t. $A \wedge B$ is unsatisfiable, *an interpolant* $\hat{A}$ is a formula in the common vocabulary of A and B s.t. $A \Rightarrow \hat{A}$ and $\hat{A} \Rightarrow \neg B$. Intuitively, $\hat{A}$ is a weakening of A that is inconsistent with B. Each node v in the infeasible counterexample is labeled by an interpolant obtained by letting A be the part of the path formula for the path from root to v, and B be the rest of the path formula. The new labeling is shown in Fig. 1 in ARG $\mathtt{A}'_1$.

Step 3. Next, the second path through `mc91` is added to $\mathtt{A}'_1$ and has to be checked for feasibility. This path has two recursive calls that need to be represented in the path formula. For each call statement, WHALE creates a new *justifying* ARG, in order to keep track of the under-approximation of the callee used in the proof of the caller and to construct the proof that the callee behaves according to a given specification.

Let $\mathtt{A}_2$ and $\mathtt{A}_3$ be the ARGs justifying the first and the second calls, respectively. For simplicity of presentation, assume that $\mathtt{A}_2$ and $\mathtt{A}_3$ have been unrolled and are identical to $\mathtt{A}_1$ in Fig. 1. The path formula π for the path 2, 5, ..., 8a is constructed by under-approximating the callees by inlining them with the justifying ARGs (shown by bold labels on the grey call edges in $\mathtt{A}'_1$). Specifically, $\pi = true \wedge (p \leq 100) \wedge (p_1 = p + 11) \wedge U_1 \wedge U_2 \wedge (r < 91)$, where U_1 and U_2 represent the under-approximations of the called functions on edges (6,7) and (7,8), respectively. This path formula is unsatisfiable and thus the counterexample is infeasible. Again, interpolants are used to strengthen node labels, as shown in ARG $\mathtt{A}''_1$. Furthermore, the interpolants are also used to generalize the under-approximations of the callees by taking the interpolant of the pair (A, B), where A is the path formula of the under-approximation and B is the rest of the path formula. The resulting interpolant $\hat{A}$ is a specification of the callee that is weaker than its under-approximation, but strong enough to exclude the infeasible counterexample. For example, to generalize the under-approximation U_1, we set A to U_1 and B to $true \wedge (p \leq 100) \wedge (p_1 = p + 11) \wedge U_2 \wedge (r < 91)$. The resulting generalizations, which happen to be $r \geq 91$ for both calls, are shown on the call edges in ARG $\mathtt{A}_1''$ with variables renamed to suit the call context.

Step 4. At this point, all intra-procedural paths of `mc91` have been examined. Hence, $\mathtt{A}_1''$ is a proof that the body of `mc91` returns $r \geq 91$ assuming that the first call returns $r \geq 91$ and that the second one returns $r \geq 91$ whenever $p \geq 91$. To discharge the assumptions, Whale sets guards and summaries for the ARGs $\mathtt{A}_2$ and $\mathtt{A}_3$ as follows: $\mathtt{G}_2 = true$, $\mathtt{S}_2 = r \geq 91$, $\mathtt{G}_3 = p \geq 91$ and $\mathtt{S}_3 = r \geq 91$, and can continue to unroll them following steps 1-3 above. However, in this example, the assumptions on recursive calls to `mc91` are weaker than what was established about the body of `mc91`. Thus, we conclude that the ARGs $\mathtt{A}_2$ and $\mathtt{A}_3$ are *covered* by $\mathtt{A}_1''$ and do not need to be expanded further, finishing the analysis. Intuitively, the termination condition is based on the Hoare proof rule for recursive functions [19] (see Sec. 3).

In practice, Whale only keeps track of guards, summaries, and labels at entry and exit nodes. Other labels can be derived from those when needed.

To summarize, Whale explores the program by unwinding its control flow graph. Each time a possible counterexample is found, it is checked for feasibility and, if needed, the labels are strengthened using interpolants. If the counterexample is inter-procedural, then an under-approximation of the callee is used for the feasibility check, and interpolants are used to guess a summary of the called function. Whale attempts to verify the summary in a similar manner, but if the verification is unsuccessful, it generates a counterexample which is used to refine the under-approximation used by the caller and to guess a new summary.

3 Preliminaries

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

Program Syntax. We divide program statements into simple statements and function calls. A *simple statement* is either an assignment statement `x = exp` or a conditional statement `assume(`Q`)`, where `x` is a program variable, and `exp` and Q are an expression and a Boolean expression over program variables, respectively. We write $[\![T]\!]$ for the standard semantics of a simple statement T.

Functions are declared as `func foo` $(p_1, \ldots, p_n) : r_1, \ldots, r_k \quad B_{\mathtt{foo}}$, defining a function with name `foo`, n parameters $\mathcal{P} = \{p_1, \ldots, p_n\}$, k return variables $\mathcal{R} = \{r_1, \ldots, r_k\}$, and body $B_{\mathtt{foo}}$. We assume that a function never modifies its parameters. The return value of a function is the valuation of all return variables at the time when the execution reaches the exit location. Functions are called using syntax $b_1, \ldots, b_k = \mathtt{foo}\ (a_1, \ldots, a_n)$, interpreted as a call to `foo`, passing *values* of local variables $a_1, \ldots, a_n$ as parameters $p_1, \ldots, p_n$, respectively, and

$$\frac{P' \Rightarrow P \ \{P\}T\{Q\} \ Q \Rightarrow Q'}{\{P'\}T\{Q'\}} \quad \frac{(P' \wedge \boldsymbol{p} = \boldsymbol{a}) \Rightarrow P \ \{P\}B_F\{Q\} \ (Q \wedge \boldsymbol{p}, \boldsymbol{r} = \boldsymbol{a}, \boldsymbol{b}) \Rightarrow Q'}{\{P'\}\boldsymbol{b} = F(\boldsymbol{a})\{Q'\}} \quad \frac{\{P\}\boldsymbol{b} = F(\boldsymbol{a})\{Q\} \vdash \{P\}B_F\{Q\}}{\{P\}\boldsymbol{b} = F(\boldsymbol{a})\{Q\}}$$

Fig. 2. Three Rules of Hoare Logic.

storing the *values* of the return variables $r_1, \ldots, r_k$ in local variables $b_1, \ldots, b_k$, respectively. The variables $\{a_i\}_{i=1}^n$ and $\{b_i\}_{i=1}^k$ are assumed to be disjoint. Moreover, for all $i, j \in [1, n]$, s.t. $i \neq j$, $a_i \neq a_j$. That is, there are no duplicate elements in $\{a_i\}_{i=1}^n$. The same holds for the set $\{b_i\}_{i=1}^k$.

Program Model. A *program* $P = (F_1, F_2, \ldots, F_n)$ is a list of n functions. Each *function* $F = (\mathcal{L}, \Delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$ is a tuple where $\mathcal{L}$ is a finite set of control locations, Δ is a finite set of actions, $\mathsf{en}, \mathsf{ex} \in \mathcal{L}$ are designated entry and exit locations, respectively, and $\mathcal{P}$, $\mathcal{R}$ and Var are sets of parameter, return and local variables, respectively (we use no global variables). An *action* $(\ell_1, T, \ell_2) \in \Delta$ is a tuple where $\ell_1, \ell_2 \in \mathcal{L}$ and T is a program statement over $\mathsf{Var} \cup \mathcal{P} \cup \mathcal{R}$. We assume that the control flow graph (CFG) represented by $(\mathcal{L}, \Delta)$ is a directed acyclic graph (DAG) (and loops are modeled by tail-recursion). Execution starts in the first function in the program. For a function $F = (\mathcal{L}, \Delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$, we write $\mathcal{L}(F)$ for $\mathcal{L}$, $\Delta(F)$ for Δ, etc. We write $\boldsymbol{p}_i$ and $\boldsymbol{r}_i$ to denote vectors of parameter and return variables of F_i.

Floyd-Hoare Logic. A *Hoare Triple* [20] $\{P\}T\{Q\}$ where T is a program statement and P and Q are propositional formulas, indicates that if P is true of program variables before executing T, and T terminates, then Q is true after T completes. P and Q are called the *pre-* and the *postcondition*, respectively.

We make use of three proof rules shown in Fig. 2. The first is the *rule of consequence*, indicating that a precondition of a statement can be strengthened whereas its postcondition can be weakened. The second is the *rule of function instantiation* where B_F is a body of a function F with parameters $\boldsymbol{p}$ and returns $\boldsymbol{r}$. It explicates the conditions under which F can be called with actual parameters $\boldsymbol{a}$, returning $\boldsymbol{b}$, and with P' and Q' as pre- and postconditions, respectively. For this rule, we assume that P is over the set of variables $\boldsymbol{p}$ and Q is over the variables $\boldsymbol{p}$ and $\boldsymbol{r}$. The third is the *rule of recursion*, indicating that a recursive function F satisfies the pre-/postconditions (P, Q) if the body of F satisfies (P, Q) *assuming* that all recursive calls satisfy (P, Q). For two sets of triples X and Y, $X \vdash Y$ indicates that Y can be proven from X (i.e., X is weaker than Y). We also say $\vdash X$ to mean that X is valid, i.e., that it follows from the axioms.

4 Inter-procedural Reachability Graphs

In this section, we introduce *Abstract Reachability Graphs* (ARGs) that extend the notion of an Abstract Reachability Tree (ART) [17] to DAGs. At a high level, an ARG represents an exploration of the state space of a function, while making assumptions about the behavior of other functions it calls. We then define a forest of ARGs, called an *Inter-procedural Abstract Reachability Graph* (iARG), to represent exploration of the state space of a program with multiple functions.

Abstract Reachability Graphs (ARGs). Let $F = (\mathcal{L}, \Delta, \mathsf{en}, \mathsf{ex}, \mathcal{P}, \mathcal{R}, \mathsf{Var})$ be a function. A *Reachability Graph* (RG) of F is a tuple $(V, E, \epsilon, \nu, \tau)$ where

- (V, E, ϵ) is a DAG rooted at $\epsilon \in V$,
- $\nu : V \to \mathcal{L}$ is a *node map*, mapping nodes to control locations s.t. $\nu(\epsilon) = \mathsf{en}$ and $\nu(v) = \mathsf{ex}$ for every leaf node v,
- and $\tau : E \to \Delta$ is an *edge map*, mapping edges to program actions s.t. for every edge $(u, v) \in E$ there exists $(\nu(u), \tau(u, v), \nu(v)) \in \Delta$.

We write $V^e = \{v \in V \mid \nu(v) = \mathsf{ex}\}$ for all leaves (*exit nodes*) in V. We call an edge e, where $\tau(e)$ is a call statement, a *call-edge*. We assume that call edges are ordered in some linearization of a topological order of (V, E).

An *Abstract Reachability Graph* (ARG) $\mathcal{A}$ of F is a tuple (U, ψ, G, S), where

- U is reachability graph of F,
- ψ is a *node labelling* that labels the root and leaves of U with formulas over program variables,
- G is a formula over $\mathcal{P}$ called a *guard*,
- and S is a formula over $\mathcal{P} \cup \mathcal{R}$ called a *summary*.

For example, ARG $\mathtt{A}_1$ is given in Fig. 1 with a guard $\mathtt{G}_1 = \mathit{true}$, a summary $\mathtt{S}_1 = r \leq 91$, and with ψ shown in braces.

An ARG $\mathcal{A}$ is *complete* iff for every path in F there is a corresponding path in $\mathcal{A}$. Specifically, $\mathcal{A}$ is complete iff every node $v \in V$ has a successor for every action $(\nu(v), T, \ell) \in \Delta$, i.e., there exists an edge $(v, w) \in E$ s.t. $\nu(w) = \ell$ and $\tau(v, w) = T$. It is *safe* iff for every leaf $v \in V$, $\psi(v) \Rightarrow S$. For example, in Fig. 2, ARG $\mathtt{A}_1''$ is safe and complete, ARG $\mathtt{A}_1'$ is complete but not safe, and other ARGs are neither safe nor complete.

Inter-procedural ARGs. An *Inter-procedural Abstract Reachability Graph* (iARG) $\mathcal{A}(P)$ of a program $P = (F_1, \ldots, F_n)$ is a tuple $(\sigma, \{\mathcal{A}_1, \ldots, \mathcal{A}_k\}, R^{\mathcal{J}}, R^{\mathcal{C}})$, where

- $\sigma : [1, k] \to [1, n]$ maps ARGs to corresponding functions, i.e., $\mathcal{A}_i$ is an ARG of $F_{\sigma(i)}$,
- $\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}$ is a set of ARGs,
- $R^{\mathcal{J}}$ is an acyclic *justification relation* between ARGs s.t. $(\{\mathcal{A}_1, \ldots, \mathcal{A}_k\}, R^{\mathcal{J}})$ is the *justification tree* of $\mathcal{A}(P)$ rooted at $\mathcal{A}_1$,
- and $R^{\mathcal{C}}$ is a *covering relation* between ARGs. Informally, if $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$ then there is a call-edge in $\mathcal{A}_i$ that is *justified* (expanded) by $\mathcal{A}_j$.

The justification tree corresponds to a partially unrolled call-graph. We write $\mathcal{A}_i \sqsubseteq_{\mathcal{J}} \mathcal{A}_j$ for the ancestor relation in the justification tree. Given two nodes $u, v \in V_i$, an inter-procedural (u, v)-path in $\mathcal{A}_i$ is a (u, v)-path in $\mathcal{A}_i$ in which every call-edge e is expanded, recursively, by a trace in an ARG $\mathcal{A}_j$, where $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$. For convenience, we assume that $\sigma(1) = 1$, and use a subscript to refer to components of an $\mathcal{A}_i$ in $\mathcal{A}(P)$, e.g., ψ_i is the node labelling of $\mathcal{A}_i$.

An ARG $\mathcal{A}_i$ is *directly covered* by $\mathcal{A}_j$ iff $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{C}}$. $\mathcal{A}_i$ is *covered* by $\mathcal{A}_j$ iff $\mathcal{A}_j \sqsubseteq_{\mathcal{J}} \mathcal{A}_i$ and $\mathcal{A}_j$ is directly covered by another ARG. $\mathcal{A}_i$ is covered iff it is

```
Require: $\mathcal{A}_i$ is uncovered and incomplete
func ExpandARG (ARG $\mathcal{A}_i$) :
    replace $U_i$ with a supergraph $U_i'$,
        where $U_i$ is the unwinding of $\mathcal{A}_i$
    Reset($\mathcal{A}_i$)
Require: $\mathcal{A}_i \not\sqsubseteq_{\mathcal{J}} \mathcal{A}_j$, $\sigma(i) = \sigma(j)$,
         $\mathcal{A}_i$ and $\mathcal{A}_j$ are uncovered,
         $\{G_j\}B_{F_{\sigma(i)}}\{S_j\} \vdash \{G_i\}B_{F_{\sigma(i)}}\{S_i\}$
func CoverARG (ARGs $\mathcal{A}_i$ and $\mathcal{A}_j$) :
    $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \setminus \{(\mathcal{A}_l, \mathcal{A}_i) \mid (\mathcal{A}_l, \mathcal{A}_i) \in R^{\mathcal{C}}\}$
    $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \cup \{(\mathcal{A}_i, \mathcal{A}_j)\}$
func Reset (ARG $\mathcal{A}_i$) :
    $\forall v \cdot \psi_i(v) \leftarrow true$
    for all $\{\mathcal{A}_j \mid \exists e \in E_i \cdot \mathcal{J}(e) = \mathcal{A}_j\}$ do
        $G_j \leftarrow true \,;\, S_j \leftarrow true$
        Reset($\mathcal{A}_j$)
func Update (ARG $\mathcal{A}_i$, $g$, $s$) :
    $G_i \leftarrow G_i \wedge g \,;\, S_i \leftarrow S_i \wedge s$
    Reset($\mathcal{A}_i$)

Require: $\mathcal{A}_i$ is uncovered, $\nu(v) = \mathsf{ex}(F_{\sigma(i)})$, $\psi_i(v) \not\Rightarrow S_i$
func RefineARG (vertex $v$ in $\mathcal{A}_i$) :
    $cond \leftarrow G_i \wedge$ iDAGCond($\mathcal{A}_i, \{v\}$) $\wedge \neg S_i$
    if $cond$ is UNSAT then
        $g_0, s_0, g_1, s_1, \ldots, s_m, s_{m+1} \leftarrow$STItp($cond$)
        $\psi_i(v) \leftarrow \psi_i(v) \wedge S_i \,;\, \psi_i(\epsilon_i) \leftarrow \psi_i(\epsilon_i) \wedge g_0$
        let $e_1, \ldots, e_m$ be topologically ordered sequence
            of all call-edges in $\mathcal{A}_i$ that can reach $v$
        for all $e_k = (u, w) \in e_1, \ldots, e_m$ do
            Update($\mathcal{J}(e_k)$, Guard($g_k$), Sum($s_k$))
    else
        if $i = 1$ then Terminate with "UNSAFE"
        $R^{\mathcal{C}} \leftarrow R^{\mathcal{C}} \setminus \{(\mathcal{A}_l, \mathcal{A}_i) \mid (\mathcal{A}_l, \mathcal{A}_i) \in R^{\mathcal{C}}\}$
        for all $\{\mathcal{A}_j \mid (\mathcal{A}_j, \mathcal{A}_i) \in R^{\mathcal{J}}\}$ do Reset($\mathcal{A}_j$)
Require: $\mathcal{A}_i$ is uncovered, safe, and complete
func UpdateGuard (ARG $\mathcal{A}_i$) :
    $G_i \leftarrow \psi(\epsilon_i)$
```

Fig. 3. The Whale Algorithm. The function STItp is used to compute interpolants and is defined later in this section.

covered by some $\mathcal{A}_j$; otherwise, it is *uncovered*. A covering relation $R^{\mathcal{C}}$ is *sound* iff for all $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{C}}$:

- $\mathcal{A}_i$ and $\mathcal{A}_j$ are mapped to the same function F_l, i.e., $\sigma(i) = \sigma(j) = l$;
- $i \neq j$ and $\mathcal{A}_i$ is not an ancestor of $\mathcal{A}_j$, i.e., $\mathcal{A}_i \not\sqsubseteq_{\mathcal{J}} \mathcal{A}_j$;
- the specification of $\mathcal{A}_j$ is stronger than that of $\mathcal{A}_i$, i.e., $\{G_j\}\boldsymbol{r} = F_l(\boldsymbol{p})\{S_j\} \vdash \{G_i\}\boldsymbol{r} = F_l(\boldsymbol{p})\{S_i\}$;
- and $\mathcal{A}_j$ is uncovered.

For example, for ARGs in Fig. 1, $(\mathtt{A}_3, \mathtt{A}_1'') \in R^{\mathcal{C}}$, and $\mathtt{A}_1''$ is uncovered. $\mathtt{A}_3$ is left incomplete, since the validity of its guard and summary follow from the validity of the guard and summary of $\mathtt{A}_1''$: $\{true\}B_{\mathtt{mc91}}\{r \geq 91\} \vdash \{p \geq 91\}B_{\mathtt{mc91}}\{r \geq 91\}$ where $(true, r \geq 91)$ and $(p \geq 91, r \geq 91)$ are the guard and summary pairs of $\mathtt{A}_1''$ and $\mathtt{A}_3$, respectively. An iARG $\mathcal{A}(P)$ is *safe* iff $\mathcal{A}_1$ is safe. It is *complete* iff every uncovered ARG $\mathcal{A}_i \in \mathcal{A}(P)$ is complete.

5 The Whale Algorithm

In this section, we provide a detailed exposition of Whale. We begin with an overview of its basic building blocks.

Overview. Given a program $P = (F_1, \ldots F_n)$ and a pair of formulas (G, S), our goal is to decide whether $\vdash \{G\}B_{F_1}\{S\}$. Whale starts with an iARG $\mathcal{A}(P) = (\sigma, \{\mathcal{A}_1\}, R^{\mathcal{J}}, R^{\mathcal{C}})$ where $\sigma(1) = 1$, and $R^{\mathcal{J}}$ and $R^{\mathcal{C}}$ are empty relations. $\mathcal{A}_1$ has one vertex v and $\nu(v) = \mathsf{en}(F_1)$. The guard G_1 and summary S_1 are set to G and S, respectively. In addition to the iARG, Whale maintains a map $\mathcal{J}$ from call-edges to ARGs and an invariant that $(\mathcal{A}_i, \mathcal{A}_j) \in R^{\mathcal{J}}$ iff there exists $e \in E_i$ s.t. $\mathcal{J}(e) = \mathcal{A}_j$.

Whale is an extension of Impact [27] to inter-procedural programs. Its three main operations (shown in Fig. 3), ExpandARG, CoverARG, and RefineARG, correspond to their counterparts of Impact. ExpandARG adds new paths to explore; CoverARG ensures that there is no unnecessary exploration, and RefineARG checks for presence of counterexamples and guesses guards and summaries. All operations maintain soundness of $R^{\mathcal{C}}$. Whale terminates either when RefineARG finds a counterexample, or when none of the operations are applicable. In the latter case, the iARG is complete. We show at the end of this section that this also establishes the desired result: $\vdash \{G_1\}B_{F_1}\{S_1\}$.

ExpandARG adds new paths to an ARG $\mathcal{A}_i$ if it is incomplete, by replacing an RG U_i with a supergraph U'_i. Implicitly, new ARGs are created to justify any new call edges, as needed, and are logged in the justification map $\mathcal{J}$. A new ARG $\mathcal{A}_j$ is initialized with a $G_j = S_j = \mathit{true}$ and $V_j = \{v\}$, where v is an entry node. The paths can be added one-at-a-time (as in Impact and in the example in Sec. 2), all-at-once (by adding a complete CFG), or in other ways. Finally, all affected labels are reset to *true*

CoverARG covers an ARG $\mathcal{A}_i$ by $\mathcal{A}_j$. Its precondition maintains the soundness of $R^{\mathcal{C}}$. Furthermore, we impose a total order, $\prec$, on ARGs s.t. $\mathcal{A}_i \sqsubset \mathcal{A}_j$ implies $\mathcal{A}_i \prec \mathcal{A}_j$, to ensure that CoverARG is not applicable indefinitely. Note that once an ARG is covered, all ARGs it covers are uncovered (line 5).

RefineARG is the core of Whale. Given an exit node v of some unsafe ARG $\mathcal{A}_i$, it checks whether there exists an inter-procedural counterexample in $\mathcal{A}(P)$, i.e., an inter-procedural (ϵ_i, v)-path that satisfies the guard G_i and violates the summary S_i. This is done using iDAGCond to construct a condition *cond* that is satisfiable iff there is a counterexample (line 16). If *cond* is SAT and $i = 1$, then there is a counterexample to $\{G_1\}\, B_{F_1}\, \{S_1\}$, and Whale terminates (line 24). If *cond* is SAT and $i \neq 1$, the guard and the summary of $\mathcal{A}_i$ are invalidated, all ARGs covered by $\mathcal{A}_i$ are uncovered, and all ARGs used to justify call edges of $\mathcal{A}_i$ are reset (lines 25-26). If *cond* is UNSAT, then there is no counterexample in the current iARG. However, since the iARG represents only a partial unrolling of the program, this does not imply that the program is safe. In this case, RefineARG uses interpolants to *guess* guards and summaries of functions called from $\mathcal{A}_i$ (lines 17-22) which can be used to replace their under-approximations without introducing new counterexamples.

The two primary distinctions between Whale and Impact are in constructing a set of formulas to represent an ARG and in using interpolants to guess function summaries from these formulas. We describe these below.

Inter-procedural DAG Condition. A *DAG condition* of an ARG $\mathcal{A}$ is a formula φ s.t. every satisfying assignment to φ corresponds to an execution through $\mathcal{A}$, and vice versa. A naive way to construct it is to take a disjunction of all the *path conditions* of the paths in the DAG. An *inter-procedural DAG condition* of an ARG $\mathcal{A}$ in an iARG $\mathcal{A}(P)$ (computed by the function iDAGCond) is a formula φ whose every satisfying assignment corresponds to an inter-procedural execution through $\mathcal{A}_i$ in $\mathcal{A}(P)$ and vice versa.

We assume that $\mathcal{A}_i$ is in Static Single Assignment (SSA) form [10] (i.e., every variable is assigned at most once on every path). iDAGCond uses the function DAGCond to compute a DAG condition[1]:

$$\begin{aligned}
&\textsc{DAGCond}(\mathcal{A}_i, X) \triangleq C \wedge D, \text{ where}\\
&\qquad C = c_{\epsilon_i} \wedge \bigwedge_{v \in V_i'} \{c_v \Rightarrow \bigvee \{c_w \mid (v,w) \in E_i\}\}\\
&\qquad D = \bigwedge_{(v,w) \in E_i'} \{(c_v \wedge c_w) \Rightarrow [\![\tau_i(v,w)]\!] \mid \tau_i(v,w) \text{ is simple}\}, \quad (1)
\end{aligned}$$

c_i are Boolean variables for nodes of $\mathcal{A}_i$ s.t. a variable c_v corresponds to node v, and $V_i' \subseteq V_i$ and $E_i' \subseteq E_i$ are sets of nodes and edges, respectively, that can reach a node in the set of exit nodes X. Intuitively, C and D encode all paths through $\mathcal{A}_i$ and the corresponding path condition, respectively. DAGCond ignores call statements which (in SSA) corresponds to replacing calls by non-deterministic assignments.

Example 1. Consider computing $\textsc{DAGCond}(\mathtt{A}_1', \{8, 8a\})$ for the ARG $\mathtt{A}_1'$ in Fig. 1, where c_8 and c_{8a} represent the two exit nodes, on the left and on the right, respectively. Then, $C = c_2 \wedge (c_2 \Rightarrow (c_3 \vee c_5)) \wedge (c_3 \Rightarrow c_8) \wedge (c_5 \Rightarrow c_6) \wedge (c_6 \Rightarrow c_7) \wedge (c_7 \Rightarrow c_{8a})$ and $D = (c_2 \wedge c_3 \Rightarrow p \leq 100) \wedge (c_3 \wedge c_8 \Rightarrow r = p - 10) \wedge (c_2 \wedge c_5 \Rightarrow p \leq 100) \wedge (c_5 \wedge c_6 \Rightarrow p_1 = p + 11)$. Any satisfying assignment to $C \wedge D$ represents an execution through 2,3,8 or 2,5,...,8, where the call statements on edges (6,7) and (7,8) set p_2 and r non-deterministically.

The function $\textsc{iDAGCond}(\mathcal{A}_i, X)$ computes an inter-procedural DAG condition for a given ARG and a set X of exit nodes of $\mathcal{A}_i$ by using DAGCond and interpreting function calls. A naive encoding is to inline every call-edge e with the justifying ARG $\mathcal{J}(e)$, but this results in a monolithic formula which hinders interpolation in the next step of RefineARG. Instead, we define it as follows:

$$\begin{aligned}
&\textsc{iDAGCond}(\mathcal{A}_i, X) \triangleq \textsc{DAGCond}(\mathcal{A}_i, X) \wedge \bigwedge_{k=1}^{m} \mu_k, \text{ where}\\
&\qquad \mu_k \triangleq (c_{v_k} \wedge c_{w_k}) \Rightarrow ((\boldsymbol{p}_{\sigma(j)}, \boldsymbol{r}_{\sigma(j)} = \boldsymbol{a}, \boldsymbol{b}) \wedge \textsc{iDAGCond}(\mathcal{A}_j, V_j^e)), \quad (2)
\end{aligned}$$

m is the number of call-edges in $\mathcal{A}_i$, $e = (v_k, w_k)$ is the kth call-edge[2], $\mathcal{A}_j = \mathcal{J}(e)$, and $\tau(e)$ is $\boldsymbol{b} = F_{\sigma(j)}(\boldsymbol{a})$. Intuitively, μ_k is the under-approximation of the kth call-edge e in $\mathcal{A}_i$ by the traces in the justifying ARG $\mathcal{A}_j = \mathcal{J}(e)$. Note that iDAGCond always terminates since the justification relation is acyclic.

Example 2. Following Example 1, $\textsc{iDAGCond}(\mathtt{A}_1', \{8, 8a\})$ is $(C \wedge D) \wedge \mu_1 \wedge \mu_2$, where $C \wedge D$ are as previously defined, and μ_1, μ_2 represent constraints on the edges $(6,7)$ and $(7,8)$. Here, $\mu_1 = (c_6 \wedge c_7) \Rightarrow ((p' = p_1 \wedge p_2 = r') \wedge$

[1] In practice, we use a more efficient encoding described in [14].

[2] Recall, call-edges are ordered in some linearization of a topological order of RG U_i.

DAGCond($\mathtt{A}_2, \{8\}$)), i.e., if an execution goes through the edge (6,7), then it has to go through the paths of $\mathtt{A}_2$ – the ARG justifying this edge. Using primed variables avoids name clashes between the locals of the caller and the callee.

Lemma 1. *Given an iARG $\mathcal{A}(P)$, an ARG $\mathcal{A}_i \in \mathcal{A}(P)$, and a set of exit nodes X, there exists a total onto map from satisfying assignments of* iDAGCond($\mathcal{A}_i, X$) *to inter-procedural (ϵ_i, X)-executions in $\mathcal{A}(P)$.* [3]

A corollary to Lemma 1 is that for any pair of formulas G and S, $G \wedge$ iDAGCond($\mathcal{A}_i, X$) $\wedge S$ is UNSAT iff there does not exist an execution in $\mathcal{A}_i$ that starts at ϵ_i in a state satisfying G and ends in a state $v \in X$ satisfying S.

Guessing Guards and Summaries. Our goal now is to show how under-approximations of callees in formulas produced by iDAGCond can be generalized. First, we define a function

$$\text{SpecCond}(\mathcal{A}_i, X, I) \triangleq \text{DAGCond}(\mathcal{A}_i, X) \wedge \bigwedge_{k=1}^{m} \mu_k,$$

where $I = \{(q_k, t_k)\}_{k=1}^{m}$ is a sequence of formulas over program variables, $\mu_k = (c_{v_k} \wedge c_{w_k}) \Rightarrow ((\boldsymbol{p}_{\sigma(j)}, \boldsymbol{r}_{\sigma(j)} = \boldsymbol{a}, \boldsymbol{b}) \wedge (q_k \Rightarrow t_k))$, and the rest is as in the definition of iDAGCond. SpecCond is similar to iDAGCond, except that it takes a sequence of pairs of formulas (pre- and postconditions) that act as specifications of the called functions on the call-edges $\{e_k\}_{k=1}^{m}$ along the paths to X in $\mathcal{A}_i$. Every satisfying assignment of SpecCond($\mathcal{A}_i, X, I$) corresponds to an execution through $\mathcal{A}_i$ ending in X, where each call-edge e_k is interpreted as $\mathtt{assume}(q_k \Rightarrow t_k)$.

Lemma 2. *Given an iARG $\mathcal{A}(P)$, an ARG $\mathcal{A}_i \in \mathcal{A}(P)$, a set of exit nodes X, and a sequence of formulas $I = \{(q_k, t_k)\}_{k=1}^{m}$, there exists a total and onto map from satisfying assignments of* SpecCond($\mathcal{A}_i, X, I$) *to (ϵ_i, X)-executions in $\mathcal{A}_i$, where each call-edge e_k is interpreted as $\mathtt{assume}(q_k \Rightarrow t_k)$.*

Given an UNSAT formula $\Phi = G_i \wedge$ iDAGCond($\mathcal{A}_i, X$) $\wedge \neg S_i$, the goal is to find a sequence of pairs of formulas $I = \{(q_k, t_k)\}_k$ s.t. $G_i \wedge$SpecCond($\mathcal{A}_i, X, I$)$\wedge$ $\neg S_i$ is UNSAT, and for every t_k, iDAGCond($\mathcal{A}_j, V_j^e$) $\Rightarrow t_k$, where $\mathcal{A}_j = \mathcal{J}(e_k)$. That is, we want to *weaken* the under-approximations of callees in Φ, while keeping Φ UNSAT. For this, we use interpolants.

We require a stronger notion of interpolants than usual: Let $\Pi = \varphi_0 \wedge \cdots \wedge \varphi_{n+1}$ be UNSAT. A sequence of formulas $g_0, s_0, \ldots, g_{n-1}, s_{n-1}, g_n$ is a *state/-transition interpolant sequence* of Π, written STItp(Π), iff:

1. $\varphi_0 \Rightarrow g_0$,
2. $\forall i \in [0, n] \cdot \varphi_{i+1} \Rightarrow s_i$,
3. $\forall i \in [0, n] \cdot (g_i \wedge s_i) \Rightarrow g_{i+1}$,
4. and $g_n \wedge \varphi_{n+1}$ is UNSAT.

[3] Proofs are available at [1]

We call g_i and s_i the state- and transition-interpolants, respectively. $\mathrm{STI}\textsc{tp}(\mathit{\Pi})$ can be computed by a repeated application of current SMT-interpolation algorithms [6] on the same resolution proof:

$$g_i = \textsc{Itp}(\bigwedge_{j=0}^{i} \varphi_j, \bigwedge_{j=i+1}^{n+1} \varphi_j, \mathit{pf}) \qquad s_i = \textsc{Itp}(\varphi_i, \bigwedge_{j=0}^{i-1} \varphi_j \wedge \bigwedge_{j=i+1}^{n+1} \varphi_j, \mathit{pf}),$$

where *pf* is a fixed resolution proof and $\textsc{Itp}(A, B, \mathit{pf})$ is a Craig interpolant of (A, B) from *pf*. The proof of correctness of the above computation is similar to that of Theorem 6.6 of [6].

Recall that RefineARG (Fig. 3), on line 16, computes a formula $\mathit{cond} = G_i \wedge \varphi \wedge \bigwedge_{k=1}^{m} \mu_k \wedge \neg S_i$ using iDAGCond for ARG $\mathcal{A}_i$ and an exit node v, where μ_k is an under-approximation representing the call-edge $e_k = (u_k, w_k)$. For simplicity of presentation, let $\tau(e_k)$ be $\boldsymbol{b}_k = F_k(\boldsymbol{a}_k)$. Assume *cond* is UNSAT and let $g_0, s_0, \ldots, s_m, g_{m+1}$ be state/transition interpolants for *cond*. By definition, each s_k is an over-approximation of μ_k that keeps *cond* UNSAT. Similarly, g_0 is an over-approximation of G_i that keeps *cond* UNSAT, and g_k, where $k \neq 0$, is an over-approximation of the executions of $\mathcal{A}_i$ assuming that all call statements on edges $e_k, \ldots, e_m$ are non-deterministic. This is due to the fact that $(G_i \wedge \varphi \wedge \mu_1 \wedge \cdots \wedge \mu_{j-1}) \Rightarrow g_j$. Note that $g_0, s_0, \ldots, s_m, g_{m+1}$ are also state/transition interpolants for the formula $G_i \wedge \varphi \wedge (g_1 \Rightarrow s_1) \wedge \cdots \wedge (g_m \Rightarrow s_m) \wedge \neg S_i$. The goal (lines 18–22) is to use the sequence $\{(g_k, s_k)\}_{k=1}^{m}$ to compute a sequence $I = \{(q_k, t_k)\}_{k=1}^{m}$ s.t. $G_i \wedge \textsc{SpecCond}(\mathcal{A}_i, \{v\}, I) \wedge \neg S_i$ is UNSAT. By definition of an interpolant, s_k is over the variables $\boldsymbol{a}_k$, $\boldsymbol{b}_k$, c_{u_k}, and c_{w_k}, whereas t_k has to be over $\boldsymbol{p}_k$ and $\boldsymbol{r}_k$, to represent a summary of F_k. Similarly, g_k is over $\boldsymbol{a}_k$, $\boldsymbol{b}_k$, c_{u_j}, and c_{w_j} for all $j \geq k$, whereas q_k has to be over $\boldsymbol{p}_k$ to represent a guard on the calling contexts. This transformation is done using the following functions:

$$\textsc{Sum}(s_k) \triangleq s_k[c_{u_k}, c_{w_k} \leftarrow \top][\boldsymbol{a}_k, \boldsymbol{b}_k \leftarrow \boldsymbol{p}_k, \boldsymbol{r}_k]$$
$$\textsc{Guard}(g_k) \triangleq \exists Q \cdot g_k[c_u \leftarrow (u_k \sqsubseteq u) \mid u \in V_i][\boldsymbol{a}_k \leftarrow \boldsymbol{p}_k],$$

where the notation $\varphi[x \leftarrow y]$ stands for a formula φ with all occurrences of x replaced by y, $w \sqsubseteq u$ means that a node u is reachable from w in $\mathcal{A}_i$, and Q is the set of all variables in g_k except for $\boldsymbol{a}_k$.

Given a transition interpolant s_k, $\textsc{Sum}(s_k)$ is an over-approximation of the set of reachable states by the paths in $\mathcal{J}(u_k, w_k)$. $\textsc{Guard}(g_k)$ sets all (and only) successor nodes of u_k to true, thus restricting g_k to executions reaching the call-edge (u_k, w_k); furthermore, all variables except for the arguments $\boldsymbol{a}_k$ are existentially quantified, effectively over-approximating the set of parameter values with which the call on (u_k, w_k) is made.

Lemma 3. *Given an ARG $\mathcal{A}_i \in \mathcal{A}(P)$, and a set of exit nodes X, let $\mathit{\Phi} = G_i \wedge \textsc{iDAGCond}(\mathcal{A}_i, X) \wedge \neg S_i$ be UNSAT and let $g_0, s_0, \ldots, s_m, g_{m+1}$ be $\mathrm{STI}\textsc{tp}(\mathit{\Phi})$. Then, $G_i \wedge \textsc{SpecCond}(\mathcal{A}_i, X, \{(\textsc{Guard}(g_k), \textsc{Sum}(s_k))\}_{k=1}^{m}) \wedge \neg S_i$ is UNSAT.*

Example 3. Let $\mathit{cond} = \mathit{true} \wedge \varphi \wedge \mu_1 \wedge \mu_2 \wedge (r < 91)$, where *true* is the guard of $\mathtt{A}'_1$, φ is $C \wedge D$ from Example 1, μ_1 and μ_2 are as defined in Example 2, and $(r <$

91) is the negation of the summary of $\mathtt{A}'_1$. A possible sequence of state/transition interpolants for *cond* is $g_0, s_0, g_1, s_1, g_2, s_2, g_3$, where $g_1 = (r < 91 \Rightarrow (c_6 \wedge c_7 \wedge c_{8a}))$, $s_1 = ((c_6 \wedge c_7) \Rightarrow p_2 \geq 91)$, $g_2 = (r < 91 \Rightarrow (c_7 \wedge c_{8a} \wedge p_2 \geq 91))$, and $s_2 = ((c_7 \wedge c_{8a}) \Rightarrow r \geq 91)$. Hence, $\textsc{Guard}(g_1) = \exists r \cdot r < 91$ (since all c_u, where node u is reachable from node 6, are set to true), $\textsc{Sum}(s_1) = r \geq 91$ (since r is the return variable of `mc91`), $\textsc{Guard}(g_2) = p \geq 91$, and $\textsc{Sum}(s_2) = r \geq 91$.

RefineARG uses $(\textsc{Guard}(g_k), \textsc{Sum}(s_k))$ of each edge e_k to strengthen the guard and summary of its justifying ARG $\mathcal{J}(e_k)$. While $\textsc{Guard}(g_k)$ may have existential quantifiers, it is not a problem for iDAGCond since existentials can be skolemized. However, its may be a problem for deciding the precondition of CoverArg. In practice, we eliminate existentials using interpolants by observing that for a complete ARG $\mathcal{A}_i$, $\psi_i(\epsilon_i)$ is a quantifier-free safe over-approximation of the guard. Once an ARG $\mathcal{A}_i$ is complete, UpdateGuard in Fig. 3 is used to update G_i with its quantifier-free over-approximation. Hence, an expensive quantifier elimination step is avoided.

Soundness and Completeness. By Lemma 1 and Lemma 2, Whale maintains an invariant that every complete, safe and uncovered ARG $\mathcal{A}_i$ means that its corresponding function satisfies its guard and summary *assuming* that all other functions satisfy the corresponding guards and summaries of all ARGs in the current iARG. Formally, let Y and Z be two sets of triples defined as follows:

$$Y \triangleq \{\{G_j\}\, \boldsymbol{b} = F_{\sigma(j)}\, (\boldsymbol{a})\{S_j\} \mid \mathcal{A}_j \in \mathcal{A}(P) \text{ is uncovered or directly covered}\}$$
$$Z \triangleq \{\{G_i\}\, B_{F_{\sigma(i)}}\, \{S_i\} \mid \mathcal{A}_i \in \mathcal{A}(P) \text{ is safe, complete, and uncovered}\}$$

Whale maintains the invariant $Y \vdash Z$. Furthermore, if the algorithm terminates, every uncovered ARG is safe and complete, and every directly covered ARG is justified by an uncovered one. This satisfies the premise of Hoare's (generalized) proof rule for mutual recursion and establishes soundness of Whale.

Whale is complete for Boolean programs, under the restriction that the three main operations are scheduled fairly (specifically, CoverARG is applied infinitely often). The key is that Whale only uses interpolants over program variables in a current scope. For Boolean programs, this bounds the number of available interpolants. Therefore, all incomplete ARGs are eventually covered.

Theorem 1. Whale *is sound. Under fair scheduling, it is also complete for Boolean programs.*

6 Implementation and Evaluation

We have built a prototype implementation of Whale using the LLVM compiler infrastructure [23] as a front-end. For satisfiability checking and interpolant generation, we use the MathSAT4 SMT solver [5]. The implementation and examples reported here are available at [1].

Our implementation of Whale is a particular heuristic determinization of the three operations described in Sec. 5: A FIFO queue is used to schedule the processing of ARGs. Initially, the queue contains only the main ARG $\mathcal{A}_1$. When

	Whale			Wolverine 0.5	Blast 2.5			
Program	**#ARGs**	**#Refine**	**Time**	**Time**	**Time (B1)**	**Time (B2)**	**#Preds (B1)**	**#Preds (B2)**
`ddd1.c`	5	3	**0.43**	4.01	4.64	1.71	15	8
`ddd2.c`	5	3	**0.59**	5.71	5.29	2.65	16	10
`ddd3.c`	6	5	**20.19**	30.56	48	20.32	25	16
`ddd1err.c`	5	1	**0.16**	3.82	0.42	1.00	25	8
`ddd2err.c`	5	1	**0.28**	5.72	0.44	0.96	5	8
`ddd3err.c`	5	11	126.4	**17.25**	TO	43.11	TO	37
`ddd4err.c`	6	1	5.73	**1.76**	24.51	CR	19	CR

Fig. 4. A comparison between Whale, Blast, and Wolverine. Time is in seconds.

an ARG is picked up from the queue, we first try to cover it with another ARG, using CoverARG. In case it is still uncovered, we apply UpdateARG and RefineARG until they are no longer applicable, or until RefineARG returns a counterexample. Every ARG created by UpdateARG or modified by Reset is added to the processing queue. Furthermore, we use several optimizations not reported here. In particular, we merge ARGs of same the function. The figures reported in this section are for the number of combined ARGs and do not represent the number of function calls considered by the analysis.

Our goal in evaluating Whale is two-fold: (1) to compare effectiveness of our interpolation-based approach against traditional predicate abstraction techniques, and (2) to compare our inter-procedural analysis against intra-procedural interpolation-based algorithms. For (1), we compared Whale with Blast [4]. For (2), we compared Whale with Wolverine [22], a recent software model checker that implements Impact algorithm [27] (it inlines functions and, thus, does not handle recursion).

For both evaluations, we used non-recursive low-level C programs written for the pacemaker grand challenge[4]. Pacemakers are devices implanted in a human's body to monitor heart rate and send electrical signals (paces) to the heart when required. We wrote test harnesses to simulate the pacemaker's interaction with the heart on one of the most complex pacemaker operation modes (DDD). The major actions of a pacemaker are sensing and pacing. Periodically, a pacemaker suspends its sensing operation and then turns it back on. The properties we checked involved verifying correct sequences of toggling sensing operations, e.g., that sensing is not suspended for more than two time steps, where we measured time steps by the number of interrupts the pacemaker receives.

Fig. 4 summarizes the results of our experiments. Blast was run in two configurations, B1 and B2[5]. Wolverine was run in its default (optimal) configuration. For Whale, we show the number of ARGs created and the number of calls to RefineARG for each program. For Blast, we show the number of predicates needed to prove or refute the property in question. 'CR' and 'TO' denote a crash and an execution taking longer than 180s, respectively. The pro-

[4] Detailed information on the pacemaker challenge is available at `http://www.cas.mcmaster.ca/wiki/index.php/Pacemaker`.

[5] B1 is `-dfs -craig 2 -predH 0` and B2 is `-msvc -nofp -dfs -tproj -cldepth 1 -predH 6 -scope -nolattice`.

grams named ddd$i$.c are safe; ddd$i$err.c have errors. While all programs are small (∼300 LOC), their control structure is relatively complex.

For example, Fig. 4 shows that Whale created five ARGs while processing ddd3.c, called RefineARG three times and proved the program's correctness in 0.59 seconds. Blast's configuration B1 tool 5.29 seconds and used 16 predicates, whereas B2 took 2.65 seconds and used 10 predicates. Wolverine's performance was comparable to B1, verifying the program in 5.71 seconds.

For most properties and programs, we observe that Whale outperforms Wolverine and Blast (in both configurations). Note that neither of the used Blast configurations could handle the entire set of programs without crashing or timing out. ddd3err.c contains a deep error, and to find it, Whale spends a considerable amount of time in SMT solver calls, refining and finding counterexamples to a summary, until the under-approximation leading to the error state is found. For this particular example, we believe Wolverine's dominance is an artifact of its search strategy. In the future, we want to experiment with heuristics for picking initial under-approximations and heuristics for refining them, in order to achieve faster convergence.

7 Related Work

The use of interpolants in verification was introduced in [25] in the context of SAT-based bounded model checking (BMC). There, McMillan used interpolation to over-approximate the set of states reachable at depth k in the model, using refutation proofs of length k BMC queries. The process continues until a counterexample is found or a fixed point is reached. At a high level, our summarization technique is similar, as we use interpolants to over-approximate the reachable states of a function by taking finite paths through it. In the context of predicate abstraction, interpolation was used as a method for deriving predicates from spurious counter-examples [18]. Interpolation was also used in [21] to approximate a program's transition relation, leading to more efficient but less precise predicate abstraction queries.

As described earlier, Whale avoids the expensive step of computing abstractions, necessary in CEGAR-based software model checking tools (e.g., Blast [17], Slam [2], and Yasm [15]). For inter-procedural verification, approaches like Slam implement a BDD-based Sharir-Pnueli-style analysis [28] for Boolean programs. It would be interesting to compare it with our SMT-based approach.

McMillan [27] proposes an intra-procedural interpolation-based software model checking algorithm, Impact, that computes interpolants from infeasible paths to an error location. Whale can be viewed as an extension of Impact to the inter-procedural case. In fact, our notion of ARG covering is analogous to McMillan's vertex covering lifted to the ARG level. While Impact unrolls loops until all vertices are covered or fully expanded (thus, an invariant is found), Whale unrolls recursive calls until all ARGs are covered or fully expanded (completed). One advantage of Whale is that it encodes all intra-procedural paths by a single SMT formula. Effectively, this results in delegating intra-procedural covering to the SMT solver.

In [26], interpolants are used as blocking conditions on infeasible symbolic execution paths and as means of computing function summaries. This approach differs from WHALE in that the exploration is not property-driven and thus is more suited for bug finding than verification. Also, handling unbounded loops and recursion requires manual addition of auxiliary variables.

Heizmann et al. [16] propose a procedure that views a program as a nested word automaton. Interpolants or predicate abstraction [12] are used to generalize infeasible paths to error and remove them from the program's automaton until no errors are reachable. In contrast to WHALE, this approach does not produce modular proofs and does not compute function summaries.

SYNERGY [13] and its inter-procedural successor SMASH [11] start with an approximate partitioning of reachable states of a given program. Partition refinement is guided by the weakest precondition computations over infeasible program paths. The main differences between WHALE and [13, 11] are: (a) interpolants focus on relevant facts and can force faster convergence than weakest preconditions [18, 26]; (b) our use of interpolants does not require an expensive quantifier elimination step employed by SMASH to produce summaries; (c) SMASH [11] does not handle recursion – in fact, our ARG covering technique can be easily adapted to the notion of queries used in [11] to extend it to recursive programs; and finally, (d) SYNERGY and SMASH use concrete test cases to guide their choice of program paths to explore. Compared to WHALE, this makes them better suited for bug finding.

8 Conclusion and Future Work

In this paper, we presented WHALE, an interpolation-based algorithm for inter-procedural verification. WHALE handles (recursive) sequential programs and produces modular safety proofs. Our key insight is the use of Craig interpolants to compute function summaries from under-approximations of functions. We showed that performance of WHALE is comparable, and often better, than state-of-the-art software model checkers from the literature.

This work opens many avenues for future research, both in terms of optimizations and extensions to other program models. For example, due to the range of interpolants that can be generated for a formula, we would like to experiment with different interpolation algorithms to test their effectiveness in this domain. We are also interested in extending WHALE to handle concurrent programs.

References

1. WHALE Homepage, http://www.cs.toronto.edu/~aws/whale
2. Ball, T., Podelski, A., Rajamani, S.: "Boolean and Cartesian Abstraction for Model Checking C Programs". In: Proc. of TACAS'01. vol. 2031, pp. 268–283 (2001)
3. Ball, T., Rajamani, S.: "The SLAM Toolkit". In: Proc. of CAV'01. LNCS, vol. 2102, pp. 260–264 (2001)
4. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: "The Software Model Checker BLAST". STTT 9(5-6), 505–525 (2007)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: "The MathSAT 4 SMT Solver". In: Proc. of CAV'08. pp. 299–303 (2008)

6. Cimatti, A., Griggio, A., Sebastiani, R.: "Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories". ACM Trans. Comput. Log. 12(1), 7 (2010)
7. 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)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
9. Craig, W.: "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory". The Journal of Symbolic Logic 22(3), 269–285 (1957)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". ACM TOPLAS 13(4), 451–490 (1991)
11. 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)
12. Graf, S., Saïdi, H.: "Construction of Abstract State Graphs with PVS". In: Proc. of CAV'97. vol. 1254, pp. 72–83 (1997)
13. 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)
14. Gurfinkel, A., Chaki, S., Sapra, S.: "Efficient Predicate Abstraction of Program Summaries". In: Proc. of NFM'11. LNCS, vol. 6617, pp. 131–145 (2011)
15. Gurfinkel, A., Wei, O., Chechik, M.: "Yasm: A Software Model-Checker for Verification and Refutation". In: Proc. of CAV'06. LNCS, vol. 4144, pp. 170–174 (2006)
16. Heizmann, M., Hoenicke, J., Podelski, A.: "Nested Interpolants". In: Proc. of POPL'10. pp. 471–482 (2010)
17. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: "Lazy Abstraction". In: Proc. of POPL'02. pp. 58–70 (2002)
18. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: "Abstractions from Proofs". In: Proc. of POPL'04. pp. 232–244 (2004)
19. Hoare, C.: "Procedures and Parameters: An Axiomatic Approach". In: Proc. of Symp. on Semantics of Algorithmic Languages. vol. 188, pp. 102–116 (1971)
20. Hoare, C.: "An Axiomatic Basis for Computer Programming". Comm. ACM 12(10), 576–580 (1969)
21. Jhala, R., McMillan, K.: "Interpoland-Based Transition Relation Approximation". In: Proc. of CAV'05. LNCS, vol. 3576, pp. 39–51 (2005)
22. Kroening, D., Weissenbacher, G.: "Interpolation-Based Software Verification with Wolverine". In: Proc. of CAV'11. LNCS, vol. 6806, pp. 573–578 (2011)
23. Lattner, C., Adve, V.: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: Proc. of CGP'04 (Mar 2004)
24. Manna, Z., McCarthy, J.: "Properties of Programs and Partial Function Logic". J. of Machine Intelligence 5 (1970)
25. McMillan, K.L.: "Interpolation and SAT-Based Model Checking". In: Proc. of CAV'03. LNCS, vol. 2725, pp. 1–13 (2003)
26. McMillan, K.: "Lazy Annotation for Program Testing and Verification". In: Proc. of CAV'10. LNCS, vol. 6174, pp. 104–118 (2010)
27. McMillan, K.L.: "Lazy Abstraction with Interpolants". In: Proc. of CAV'06. LNCS, vol. 4144, pp. 123–136 (2006)
28. Sharir, M., Pnueli, A.: Program Flow Analysis: Theory and Applications, chap. "Two Approaches to Interprocedural Data Flow Analysis", pp. 189–233. Prentice-Hall (1981)