Unbounded Procedure Summaries from Bounded Environments

Lauren Pick[1], Grigory Fedyukovich[2][0000−0003−1727−4043], and Aarti Gupta[1][0000−0001−6676−9400]

[1] Princeton University, Princeton NJ, USA
[2] Florida State University, Tallahassee, FL, USA

Abstract. Modular approaches to verifying interprocedural programs involve learning summaries for individual procedures rather than verifying a monolithic program. Modern approaches based on use of Satisfiability Modulo Theory (SMT) solvers have made much progress in this direction. However, it is still challenging to handle mutual recursion and to derive adequate procedure summaries using scalable methods. We propose a novel modular verification algorithm that addresses these challenges by learning lemmas about the relationships among procedure summaries and by using *bounded environments* in SMT queries. We have implemented our algorithm in a tool called Clover and report on a detailed evaluation that shows that it outperforms existing automated tools on benchmark programs with mutual recursion while being competitive on standard benchmarks.

Keywords: Program verification · modular verification · procedure summaries · bounded environments · CHC solvers.

1 Introduction

Automated techniques for modular reasoning about interprocedural recursive programs have a rich history with various techniques spanning interprocedural dataflow analysis [56, 54], abstract interpretation [18], and software model checking [6]. These techniques exploit the inherent modularity in a program by deriving a *summary* for each procedure. Procedure summaries can be viewed as specifications or interface contracts, where internal implementation details have been abstracted away. In addition to aiding code understanding and maintenance, they can be combined to verify the full program. A modular verification approach that infers and composes procedure summaries may scale better than a monolithic one that considers all procedure implementations at once.

A popular modern approach is to encode interprocedural program verification problems as Constrained Horn Clauses (CHCs) [32], in which uninterpreted predicates represent placeholders for procedure summaries. A CHC solver then finds interpretations for these predicates such that these interpretations correspond to summaries, enabling generation of procedure summaries.

CHC solvers [32, 49, 42, 13, 39, 59, 27] query to backend SMT (Satisfiability Modulo Theory) solvers [8] to find interpretations that make all CHC rules valid. In addition to classic fixpoint computations, CHC solvers use model checking techniques, e.g., counterexample guided abstraction refinement (CEGAR) [17], interpolation [46], property-directed reachability (PDR) [12, 23], and guess-and-check procedures [25]. They can thus find procedure summaries adequate for verification but not necessarily least or greatest fixpoints. CHC-based verifiers have been successfully applied to a range benchmark programs, but there remain significant challenges in handling mutual recursion and in scalability.

We aim to address these challenges by leveraging program structure during solving and learning relevant facts. Typical CHC-based verifiers may not maintain a program's structure when encoding it into CHCs. In contrast, our method uses the program call graph, which can be preserved easily in a CHC encoding, to guide proof search.

For improving scalability, we ensure that the *SMT queries in our method are always bounded in size* even when more of the program is explored. We wish both to maintain scalability *and* to avoid learning over-specialized facts. We do this by leveraging the call graph of the program, i.e., analyzing a procedure in the context of a bounded number of levels in the call graph. Furthermore, such a notion of a *bounded environment* enables us to refer to bounded call paths in the program and learn special lemmas, called *EC (Environment-Call) Lemmas*, to capture relationships among summaries of different procedures on such paths. These lemmas are beneficial in *handling mutual recursion*.

Other techniques also trade off scalability and relevance by considering a bounded number of levels in a call graph, e.g., in bounded context-sensitivity or k-sensitive pointer/alias analysis [51], stratified inlining [44], and depth cut-off [40] in program verification. However, other than Spacer [42], which is restricted to $k = 1$ bounded environments, existing CHC solvers do not use bounded environments to limit size of the SMT queries.

Summary of Contributions. This paper's contributions are as follows:

- We propose a new CHC-solving method for generating procedure summaries for recursive interprocedural programs (§6).
- We propose to handle mutual recursion by explicitly learning EC Lemmas to capture relationships among different procedures on a call path (§5).
- We propose to use bounded environments (with bound $k \geq 1$) (§4) to compute individual procedure summaries. The SMT queries formulated in our method are always bounded in size, thereby improving scalability.
- We have implemented our method in a tool called Clover and report on its evaluation on several benchmark programs, along with a detailed comparison against existing tools (§7).

To the best of our knowledge, EC Lemmas and bounded environments, the main features of our algorithm, are novel for summary generation in modular verifiers.

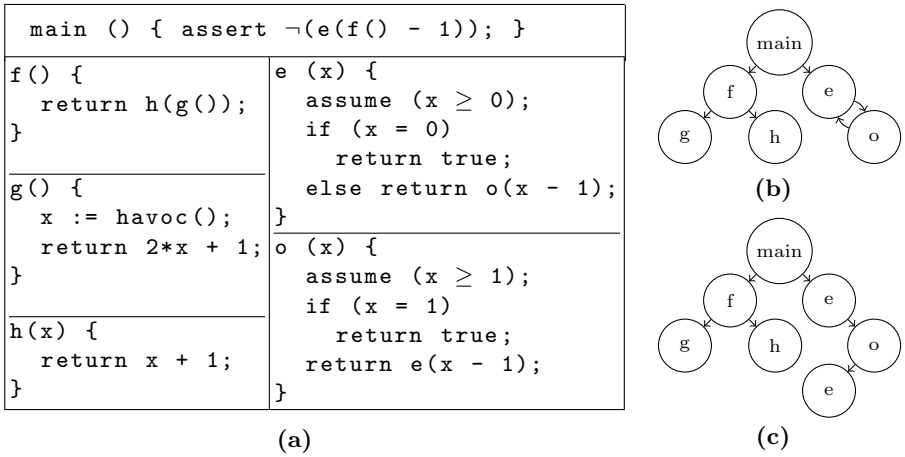


Fig. 1: Example: (a) source code, (b) call graph, and (c) final derivation tree.

2 Motivating Example

We illustrate the main steps of our modular algorithm on the example program shown in Fig. 1a. To keep our focus on intuition, we describe our algorithm in terms of the program (CHC encodings are described later).

In Fig. 1a, `e` and `o` are defined mutually recursively and return *true* iff their argument is respectively even or odd. Procedure `f` returns the (always-even) result of calling `h` on `g`'s result, where `g` returns an arbitrary odd number and `h` adds one to its input. The safety specification is that $\texttt{e}(\texttt{f}() - 1)$ never holds. We aim to infer over-approximate procedure summaries so that the assertion's truth follows from replacing procedure calls in `main` with these summaries.

We maintain context-insensitive *over-* and *under-approximate* summaries for all procedures, each of which captures both pre- *and* post-conditions of its procedure. All over- (resp. under-) approximate summaries are initially $\top$ (resp. $\bot$). At each step, we choose a target procedure p and its bounded environment, then update p's summaries based on the results of SMT queries on its over- or under-approximate body. We also allow the bounded environment to be over- or under-approximated, leading to four kinds of SMT queries. These queries let us over- and under-approximate any procedure that is called before or after the target, unlike Spacer [42] or Smash [31], which use two kinds of SMT queries.

Table 1 lists non-trivial verification steps that update various procedure summaries. (Steps that do not update any summary are not listed.) The first column lists the call path that is visited in each step, in which the last call is the current *target* procedure whose summary is updated, and the call path is used to generate its bounded environment. The "Environment" (resp. "Target") column shows whether the bounded environment (resp. target) is over- or under-approximated. The "Deductions" column lists deductions resulting from SMT queries in that step. Note that formulas in this column (and in the remainder of this section) are implicitly universally quantified over all variables and involve uninterpreted

Table 1: Relevant steps to verify program in Fig. 1a.

	Call graph path	Environment	Target	Deductions (universally quantified)
1	`main` → `e`	Over	Under	$x = 0 \wedge y = \mathit{true} \Rightarrow e(x, y)$
2	`main` → `f` → `h`	Over	Under	$y = x + 1 \Rightarrow h(x, y)$
3	`main` → `e` → `o`	Over	Under	$x = 1 \wedge y = \mathit{true} \Rightarrow o(x, y)$
4	`main` → `f` → `g`	Over	Under	$y \bmod 2 \neq 0 \Rightarrow g(y)$
5	`main` → `f` → `g`	Under	Over	$g(y) \Rightarrow y \bmod 2 \neq 0$
6	`main` → `f` → `h`	Under	Over	$h(x, y) \Rightarrow y = x + 1$
7	`main` → `f`	Over	Under	$y \bmod 2 = 0 \Rightarrow f(y)$
8	`main` → `f`	Under	Over	$f(y) \Rightarrow y \bmod 2 = 0$
9	`main` → `e` → `o` → `e`	Over	Over	$(o(x, y) \Rightarrow y \Leftrightarrow ((1 + x) \bmod 2 = 0)) \Rightarrow$ $(e(m, n) \Rightarrow n \Leftrightarrow (m \bmod 2 = 0))$
10	`main` → `e` → `o`	Over	Over	$o(x, y) \Rightarrow y \Leftrightarrow ((1 + x) \bmod 2 = 0))$ $e(x, y) \Rightarrow x > 1 \wedge y \Leftrightarrow ((1 + x) \bmod 2 = 0)$

predicates (e.g., `h(x, y)` in row 2). Except in row 9, all these formulas are implications that represent procedure summaries. Row 9 shows an implication between two such formulas – this is an instance of an EC lemma (described later).

2.1 Using the Program Call Graph

Our algorithm chooses environment-target pairs based on the call graph of the program, shown in Fig. 1b. It maintains explored paths through the call graph in a data structure called a *derivation tree*, initially consisting of only one node that represents entry procedure `main`. Fig. 1c shows the tree just before the algorithm converges. The subset A of nodes *available* to be explored is also maintained, and it is this subset that guides exploration in our algorithm.

To improve scalability, we use *bounded* environments from call paths to use in SMT queries at each step. These bounded environments include bodies of the ancestors of the target procedure, but only up to level k above the target in the call graph. Ancestors at $l > k$ above the target are soundly abstracted away so that these environments capture at least the behaviors of the program before and after the target procedure that may lead to a counterexample. Approximations of these environments and of the bodies of target procedures help us learn new facts about the targets. In this example, we use $k = 2$. When we target the last call to `e` along path `main` → `e` → `o` → `e`, `main`'s body will be abstracted.

2.2 Summary Updates using SMT Queries

We now consider the four SMT queries on a chosen environment-target pair at each step. Suppose our algorithm has already considered the path to `o` on row 3 (Table 1) and now chooses node `g` $\in A$ in path `main` → `f` → `g`. Here, the bounded environment includes calls to `h` (called by `f`) and `e` (called by `main`), so we use their over-approximate summaries (both currently $\top$). We under-approximate the environment using summaries for `h` and `e` learned in rows 2 and 1, respectively. Over- and under-approximations of `g` are just its body with local variables rewritten away (i.e., `2 * havoc() + 1`), since it has no callees.

In checks that over-approximate the procedure body, we try to learn an interpolant that proves the absence of a subset of counterexamples along this path in the program. Since target procedure body is over-approximated, any interpolant found that separates its encoding from the counterexample captured by the environment will be an over-approximate summary for the target procedure, expressing a fact about all behaviors of the procedure. Such over-approximate summaries allow us to prove safety in a modular way. In checks that under-approximate the procedure body, we try to find (part of) a bug. Since the target procedure body is under-approximated, the interpolant is instead an under-approximate summary, describing behaviors the procedure *may* exhibit. Under-approximate summaries allow us to construct counterexamples in the case where the program is unsafe. Approximating the environment and target procedure body allows us to keep queries small.

Both the *over-over* and *under-under* checks fail here, so no updates are made. A weaker version of the *under-under* check is the *over-under* check, in which the environment is now *over-approximated*. Because it is weaker, it may result in learning under-approximate summaries that may not be necessary, since the over-approximated environment may contain spurious counterexamples. When our algorithm performs this check, it finds a path that goes through the over-approximated environment and the under-approximation of `g`'s body and thus augments `g`'s under-approximation (row 4).

A corresponding weaker version of the *over-over* check is the *under-over* check, in which the environment is *under-approximated*. Because the under-approximated environment may not capture all counterexamples, the learned interpolant by itself could be too weak to prove safety. Our algorithm refines `g`'s over-approximation with the interpolant learned in this query (row 5).

Note that these two weaker checks are crucial in our algorithm. Consider a different `main` function that contains only `assert(f() mod 2 = 0)`. To prove safety, we would need to consider paths `main` $\rightarrow$ `f` $\rightarrow$ `h` and `main` $\rightarrow$ `f` $\rightarrow$ `g`, but for these paths, both "stronger" checks fail. Paths through the derivation tree must be paths through the call graph, so we would not consider the bodies of `h` and `g` simultaneously; the "weaker" checks allow us to learn summary updates.

2.3 Explicit Induction and EC lemmas

To demonstrate the need for and use of induction and EC lemmas for handling mutual recursion, we now consider row 9 in Table 1, where we perform an *over-over* check for the final call to `e` in the call path. The current derivation tree has the same structure as the final derivation tree, shown in Fig. 1c.

No induction. At this stage, our over-approximation for `f` precisely describes all possible behaviors of `f` (rows 7, 8), but no interpolant can be learned because the over-approximation $\top$ of `o` in the body of procedure `e` is too coarse. Without using induction, we cannot make any assumptions about this call to `o`, and are stuck with this coarse over-approximation. Even if we inlined the body of `o`, we would similarly still have an overly-coarse over-approximation for `e`.

Induction with EC lemmas. We can instead try to use induction on the body of `e`. Its over-approximated environment includes counterexample paths that we

would like to prove spurious. Let formula $\phi(x, y)$ denote property $\mathtt{e}(x, y) \Rightarrow (y \Leftrightarrow x \bmod 2 = 0)$. The consequent in this implication is generated by examining the environment for `e`, i.e., the environment implies the negation of the consequent[3]. Problems arise when trying to prove this property by induction because there is no opportunity to apply the inductive hypothesis about `e`. When the else branch is taken, facts about `o` are needed to finish the proof for $\phi(x, y)$ and $x > 0$.

If we were to inline `o` and assume inductive hypothesis that $\phi(x, y)$ holds for the inner call to `e`, an inductive proof would succeed without using EC lemmas. However, such an inlining approach can lead to poor scalability and precludes inference of summaries (e.g., for `o`) that could be useful in other call paths.

EC lemmas. Our algorithm discovers additional lemmas in the form of implications over certain procedure summaries (§5). Let formula $\theta(m, n) \stackrel{\text{def}}{=} \mathtt{o}(m, n) \Rightarrow (n \Leftrightarrow (1+m) \bmod 2 = 0)$. (Again, the consequent in this implication is generated by examining the environment for `o`.) Let $\psi(x, y, m, n) \stackrel{\text{def}}{=} \theta(m, n) \Rightarrow \phi(x, y)$, i.e., ψ is similar to ϕ property, but with an additional assumption θ about `o`.

Validity of ψ is proved by case analysis: $\psi(1, \mathit{true}, m, n)$ is trivially true, and the proof of $\psi(x, y, m, n)$ for $x > 0$ works because of the assumption θ. Thus, the formula $\psi(x, y, m, n)$ is learned as an *EC lemma* (see row 9).

Now, we reconsider the call to `o` along call path $\mathtt{main} \rightarrow \mathtt{e} \rightarrow \mathtt{o}$. The discovered EC lemma allows us to prove formula θ valid by induction. This new over-approximate fact for `o` is combined with the EC lemma allowing the algorithm to learn $\mathtt{e}(x, y) \Rightarrow (y \Leftrightarrow x \bmod 2 = 0)$. This step corresponds to row 10.

3 Preliminaries

In this section, we define our notion of a program, introduce CHC notions and encodings, and define contexts and derivation trees.

Programs. A program P is a set of procedures with entry point *main*. Each procedure p has vectors of input and output variables in_p and out_p and a body body_p, which may contain calls to other procedures or recursive calls to p. When p is clear from context, we omit it in the variables' subscripts, e.g., $p(\mathit{in}, \mathit{out})$. We encode a program as a system of CHCs $\mathcal{C}$.

Definition 1. *A CHC C is an implicitly universally-quantified implication formula in first-order logic with theories of the form body $\Rightarrow$ head. Let $\mathcal{R}$ be a set of uninterpreted predicates. The formula head may take either the form $R(\vec{y})$ for $R \in \mathcal{R}$ or $\bot$. Implications in which head $= \bot$ are called* queries. *The formula body may take the form $\phi(\vec{x})$ or $\phi(\vec{x}) \wedge R_0(\vec{x}_0) \wedge \ldots \wedge R_n(\vec{x}_n)$, where each R_i is an uninterpreted predicate, and $\phi(\vec{x})$ is a fully interpreted formula over $\vec{x}$ (i.e., it contains only theory predicates), which may contain all variables in each $\vec{x}_i$ and (if the head is of the form $R(\vec{y})$) all variables in $\vec{y}$.*

[3] Expressions such as $x \bmod 2 = 0$ can be generated by existentially quantifying local variables and then performing quantifier elimination.

Body of `main`:	CHC for `main`:
`assert (¬(e(f() - 1)));`	$f(x) \wedge e(x-1,y) \wedge y \Rightarrow \bot$
Body of `e`:	**CHC for `even`:**
`assume (x ≥ 0);`	
`if (x = 0) return true;`	$x = 0 \Rightarrow e(x, \top)$
`else return o(x - 1);`	$x \neq 0 \wedge o(x-1,y) \Rightarrow e(x,y)$
Unfolding `e` in `main`:	**Unfolding `e` in `main` (CHCs):**
`assume(f() − 1 ≥ 0);`	$f(x) \wedge x - 1 = 0 \Rightarrow \bot$
`if (f() - 1 = 0) return true;`	$f(x) \wedge x - 1 \geq 0 \wedge x - 1 \neq 0 \wedge o(x-2,y) \Rightarrow \bot$
`else return o(f() − 2);`	

Fig. 2: Unfoldings (and intermediate steps) of `e` in the body of `main` from Fig. 1a. Program snippets are shown on the left and CHC encodings on the right.

A system of CHCs for a program can be generated by introducing an uninterpreted predicate per procedure and encoding the semantics of each procedure using these and theory predicates. Each application $R(\vec{x})$ in the body of a CHC corresponds to a procedure call to a procedure p, where $\vec{y} = (in_p, out_p)$. By analogy, we refer each such R as a *callee* of the predicate in the head of the CHC. For each $C \in \mathcal{C}$ with uninterpreted predicate applications $\{R_0(\vec{x}_0), \ldots, R_n(\vec{x}_n)\}$ in its body, we let $callee_C$ be a one-to-one mapping from $0, \ldots, n$ to these applications. This mapping allows us to distinguish between different applications of the same predicate within the same CHC body, which we can understand as distinguishing between different callsites of the same callee within a procedure. We abuse notation and denote the corresponding predicate for a procedure $p \in P$ in encoding $\mathcal{C}$ as p. We assume that in any application $p(\vec{y})$ in the head of a CHC in $\mathcal{C}$, $\vec{y}$ is the same vector of variables in_p, out_p. We let $C.body$ and $C.head$ denote the body and head of CHC C respectively. We let loc_C denote $fv(C.body) \setminus fv(C.head)$, where for a formula F, $fv(F)$ denotes the free variables in F. We assume that all $C, C' \in \mathcal{C}$ are such that $loc_C \cap loc_{C'} = \emptyset$ and let $loc_p = \bigcup\{loc_C \mid C.head = p(\vec{y})\}$. Note that disjunction $\bigvee_i \{body_i \mid body_i \Rightarrow p(\vec{y}) \in \mathcal{C}\}$ gives the semantics of $body_p$. We abuse notation to use $body_p$ to refer to this disjunction.

Corresponding CHC encodings are shown in Fig. 2 for demonstration. We assume the use of an encoding that preserves the call graph structure of the program in CHCs; i.e., there will be a CHC with head containing p with an application of q in its body iff p calls q.

Definition 2 (Solution). *A* solution *for a system of CHCs $\mathcal{C}$ is an interpretation M for uninterpreted predicates $\mathcal{R}$ that makes all CHCs in $\mathcal{C}$ valid.*

A CHC encoding is such that if it has a solution, the original program is safe. To remember facts learned during our algorithm, we maintain two sets of first-order interpretations of $\mathcal{R}$ called O and U, functioning as mappings from from procedures to their over- and under-approximate summaries, respectively.

Definition 3 (Procedure Summaries). *The over- (O) and under-approximate (U) summaries are such that all non-query CHCs body $\Rightarrow$ head $\in \mathcal{C}$ are valid under O and that implication head $\Rightarrow$ body is valid under U.*

From Def. 3, it is clear that for all p, $O[p] = \top$ and $U[p] = \bot$ are valid summaries. We use these as *initial* summaries in the algorithm presented in Sect. 6. Note that when O is a solution for the system of CHCs $\mathcal{C}$ (i.e., O makes the query CHCs valid). When U is such that a query CHC is *not* valid, then verification fails and a counterexample exists.

Definition 4 (Approximation). *Given a formula Π and an interpretation $M \in \{O, U\}$, an approximation $\widehat{\Pi}_M$ is defined as follows:*

$$\widehat{\Pi}_M \stackrel{\text{def}}{=} \Pi \wedge \bigwedge_{p(in,out) \text{ in } \Pi} M[p](in, out)$$

In addition to approximations, we can manipulate CHCs using *renaming* and *unfolding*.

Definition 5 (Renaming). *For a formula F containing variables $\vec{x}$, $F[\vec{x} \mapsto \vec{y}]$ denotes the simultaneous renaming of variables $\vec{x}$ to $\vec{y}$ in F.*

Definition 6 (Unfolding). *Let $\mathcal{C}$ be a system of CHCs. Let $C \in \mathcal{C}$ be a CHC $R_0(\vec{x}_0) \wedge \ldots \wedge R_n(\vec{x}_n) \wedge \phi(\vec{x}) \Rightarrow R(\vec{y})$ where $callee_C(i) = R_i(\vec{x}_i)$ for each $i \in \{0, \ldots, n\}$. There is an unfolding of $callee_C(k)$ per CHC in $\mathcal{C}$ whose head is an application of predicate R_k. For such a CHC body $\Rightarrow R_k(\vec{y}_k) \in \mathcal{C}$, the unfolding of $R_k(\vec{x}_k)$ in C is given by the following:*

$$\bigwedge_{i \in \{0,\ldots,n\}, i \neq k} R_i(\vec{x}_i) \wedge body[\vec{y}_k \mapsto \vec{x}_k] \wedge \phi(\vec{x}) \Rightarrow (\vec{y})$$

An unfolding is essentially a one-level inlining of one CHC in another. Fig. 2 illustrates what an unfolding of CHCs correponds to on our motivating example, where `e` is unfolded in `main`.

Definition 7 (Environment). *For a CHC C of the form $\bigwedge_{i \in \{1..n\}} R_i(\vec{x_i}) \wedge \phi(\vec{x_1}, \ldots, \vec{x_n}) \Rightarrow \bot$, the environment for $R_k(\vec{x_k})$ is given by the following:*

$$\bigwedge_{i \in \{1..n\}, i \neq k} R_i(\vec{x_i}) \wedge \phi(\vec{x_1}, \ldots, \vec{x_n})$$

By analogy with programs, the environment for $R_k(\vec{x_k})$ intuitively captures the procedure calls in C before and after the procedure call for $R_k(\vec{x_k})$. Note that if C is simply an encoding of a single procedure body, then the environment will only capture the immediate callees of that procedure. On the other hand, if C is, for example, an unfolding of the CHC representing `main`, then the environment may contain any calls before and after the call corresponding to $R_k(\vec{x_k})$ in a full but potentially spurious counterexample run of the program so long as they have corresponding predicate applications in the unfolding C.

Definition 8 (Derivation Tree). *A derivation tree $D = \langle N, E \rangle$ for system of CHCs P is a tree with nodes N and edges E, where each $n \in N$ is labeled with uninterpreted predicate $p = proc(n)$, a* context *query CHC $ctx(n)$, and an index $i = idx(n)$ such that $callee_{ctx(n)}(i)$ is an application of p.*

Our algorithm uses the derivation tree is capture the already-explored unfoldings starting from the encoding of *main* and to further guide exploration. Each node $n \in N$ represents a *verification subtask*, where the body of $ctx(n)$ represents a set of (potentially spurious) counterexamples. The goal of each subtask is to find a solution for the system of CHCs consisting of all non-query CHCs in $\mathcal{C}$ with the query CHC $ctx(n)$ and refine the over-approximation $O[proc(n)]$ to reflect the learned facts, or, if this cannot be done, to expand $proc(n)$'s under-approximation $U[proc(n)]$ to demonstrate (part of) a real counterexample.

A program's initial derivation tree consists of only one node labeled with procedure *main* and a query CHC from the system $\mathcal{C}$. We maintain the invariant that if s is the parent of t, then the $ctx(t)$ must be able to be constructed by unfolding a predicate in $ctx(s)$. Furthermore, we require that the unfolded predicate is one of the predicates that was added in the previous unfolding step to get $ctx(s)$. This notion of a derivation tree is similar to other CHC-based work [49, 59], but our invariant restricts the way in which we can expand the tree (i.e., the way in which we can unfold from *main*) – *every derivation tree path corresponds to a call graph path.* We let $e(n)$ refer to the environment for $callee_{ctx(n)}(idx(n))$ in $ctx(n)$.

For a derivation tree path d (of length $|d|$) whose final node is n, the *full* context $ctx(n)$ can be derived by unfolding all of $proc(n)$'s ancestors in the root node's context CHC along the corresponding call graph path for the original program[4]. We also denote this full context as $unfold(d, |d|)$. For $k < |d|$, $unfold(d, k)$ corresponds to unfolding the bodies of the last $k - 1$ procedure calls in d into the body of $proc(n)$'s k^{th} ancestor. Note that $unfold(d, k)$ only unfolds ancestors on the call path; any other of the ancestors remain represented as uninterpreted predicates. For $k \geq |d|$, $unfold(d, k) = unfold(d, |d|)$. (See also Def. 9.)

4 Bounded Contexts and Environments

Here we define *bounded* contexts and environments. Our algorithm uses these bounded versions in all SMT queries described later.

Definition 9 (Bounded context). *For a given bound k, and a path $d = n_0 \to \ldots \to n_{m-1} \to n_m$ in a derivation tree, a* k-bounded context *for n_m is a formula $bctx(n_m)$ over variables $bvs \stackrel{\text{def}}{=} fv(unfold(d, k))$, defined as follows:*

$$bctx(n_m) \stackrel{\text{def}}{=} unfold(d, k).body \wedge interface(d, k) \wedge summ(d, k) \Rightarrow \bot$$

Here, we also have the following:

- *$interface(d, k)$ is a formula over the inputs and outputs of the procedure for node n_{m-k}, $k < m$ (or $\top$, if $k \geq m$)*

[4] We lift the ancestor relationship from nodes to their procedures.

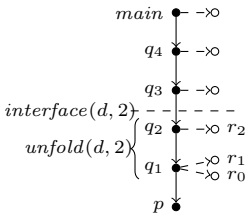
– *summ*(d, k) *is a formula over the inputs and outputs of the other callees of the* k*-bounded ancestors of* $proc(n_m)$*.*

Note $unfold(d, k)$ ignores any restrictions due to ancestors that are more than k-levels above $proc(n_m)$. Such restrictions are expressed in $interface(d, k)$, which represents the interface between the k-bounded context and the rest of the context above it. In practice, we compute $interface(d, k)$ as $\text{QE}(\exists fv(e(n_m)) \setminus \widehat{bvs.e(n_m)}_{O,\ell})$, where QE denotes quantifier elimination. We approximate quantifier elimination using the standard model-based projection technique [10]. We can always use $interface(d, k) = \top$, which treats ancestor procedures above bound k as havocs; we found this choice ineffective in practice.

In what follows, we refer to $unfold(d, k).body \wedge interface(d, k)$ as $B(d, k)$ or simply as B when d and k are clear. Again, we require that each $bctx(n_m)$ (and thus each $B(d, k)$) can be computed from its parent n_{m-1}'s bounded context via a single unfolding. Given our choice of $interface(d, k)$, using such a method to compute a child node's bounded context lets us avoid (approximate) quantifier elimination on large formulas since only one procedure body's variables need to be eliminated when starting from the parent's bounded context.

The $summ(d, k)$ formula can be either $\top$ or a conjunction that adds approximation constraints based on summaries for the other callee procedures. We use $bctx.body = B$ when $summ(d, k) = \top$, or $bctx.body = \widehat{B}_M$ or $bctx.body = \widehat{b}_M$ for $M \in \{O, U\}$, where b is the environment for $callee_{ctx(n_m)}(idx(n_m))$ (when $summ(d, k)$ is the conjunction from approximating with M).

Example 1. The figure shows a bounded context for predicate p with bound 2 for the derivation tree path shown with solid edges. Ancestor predicates q_1, q_2 are unfolded in $unfold(d, 2)$, and $summ(d, 2)$ approximates callees r_0, r_1, r_2:



For scalability, our algorithm (§6) considers verification subtasks with the *bounded* context of a given procedure. Our algorithm' queries use *bounded environments*, which can be computed from bounded contexts.

Definition 10 (Bounded environment). *For a node* n*, its bounded environment* $benv(n)$ *is the environment for the predicate* $callee_{ctx(n)}(idx(n))$ *in* $ctx(n)$*.*

We define a bounded parent relationship between nodes $s, t \in N$, where $s \to t$ is not necessarily in E, but where $ctx(s)$ has $proc(t)$ as a callee.

Definition 11 (Bounded parent). *A node* s *is a* bounded parent *of node* t *in derivation tree* D*, denoted* $s \in Bparent(t, D)$*, iff there is some index* i *such that* $callee_{ctx(s)}(i)$ *is an application of* $proc(t)$ *and* $b_t \Leftrightarrow next(b_s, proc(t), i)$*, where* b_t *and* b_s *are bodies of the bounded contexts of* s *and* t*.*

Note that the parent of a node n is always a bounded parent for n, and that n may have several bounded parents because the approximation of different full environments may lead to the same bounded environment. We use bounded parents in our algorithm (§6) to avoid considering redundant verification subtasks.

5 EC lemmas

We also learn a set L of *EC lemmas*, which are implications capturing assumptions under which a procedure has a particular over-approximation.

Definition 12 (Environment-Call (EC) Lemmas). *Let $proc(n) = p$ for some node n in a derivation tree. An EC lemma for p, where n has ancestors with procedures $\{q_i\}$ along a derivation tree path, is of the following form:*

$$\forall fv(S_i) \ \cup \ in \ \cup \ out. \bigwedge_i S_i \Rightarrow (p(in, out) \Rightarrow prop)$$

Here, $prop$ is a formula with $fv(prop) \subseteq in \ \cup \ out$, each S_i is of the form $q_i(in_i, out_i) \Rightarrow prop_i$, where q_i is some ancestor's uninterpreted predicate, and $prop_i$ is a formula with $fv(prop_i) \subseteq in_i \cup out_i$.

Intuitively, an EC lemma allows us to learn that *prop* is an over-approximation of procedure p under the assumptions $\{S_i\}$ about its ancestors with procedures $\{q_i\}$. Each S_i itself is an assumption that $prop_i$ over-approximates q_i. These ancestors are in target p's environment, so we call these formulas Environment-Call (EC) Lemmas. In practice, we learn EC lemmas involving ancestors whose procedures are callees of p to help set up induction for mutual recursion.

6 Modular Verification Algorithm

We now describe our modular verification algorithm. We first outline the top-level procedure (§6.1) based on iteratively processing nodes in the derivation tree. Then we describe how each node is processed using SMT queries (§6.2), the order in which SMT queries are performed (§6.3), and how induction is performed and how EC lemmas are learned and used (§6.4). We present the correctness and the progress property of our algorithm and discuss limitations (§6.5). (Additional heuristics are described in Appendix B.)

6.1 Algorithm Outline

Our algorithm constructs a derivation tree based on the call graph of the program, which is used to guide the selection of CHCs to explore. We *achieve scalability* by considering only bounded environments in all our SMT queries. We present these queries as part of proof rules that capture the major steps of our algorithm. The use of induction and EC lemmas enables handling of *mutually recursive programs*. The state during verification is captured by *proof (sub)goals*.

Definition 13 (Proof (sub)Goal). *For system of CHCs $\mathcal{C}$, derivation tree $D = \langle N, E \rangle$, a subset $A \subseteq N$ of* available *nodes, over- and under-approximate summary maps O and U, a set of EC lemmas L, and $Res \in \{\top, \bot\}$, a* proof (sub)subgoal *is denoted $D, A, O, U, L, \mathcal{C} \vdash Res$.*

Algorithm 1 Modular Verification Procedure

```
1: procedure VERIFY( )
2:     N ← {n} with proc(n) = main
3:     Goal ← ⟨N, ∅⟩, N, O, U, ∅, C ⊢ Res
4:     while Goal.A ≠ ∅ or summaries are insufficient do
5:         Goal ← PROCESSNODE(n, Goal) for n ∈ Goal.A
6:     return RESULT(Goal)
```

Main loop. Alg. 1 shows the top-level procedure for our method. The VERIFY procedure constructs an initial proof goal containing an initial derivation tree, initial summary maps, and empty sets of lemmas. Initially all nodes in the derivation tree are available, i.e., they are in A. It then iteratively chooses an available node and tries to update its summaries (using routine PROCESSNODE), thereby updating the current goal. The loop terminates when no more nodes are available or when the current summaries are sufficient to prove/disprove safety. RESULT returns *safe* if the summaries are sufficient for proving safety, *unsafe* if they are sufficient for disproving safety, or *unknown* otherwise.

Choice of procedures and environments. PROCESSNODE can be viewed as making queries on an environment-procedure pair. If the algorithm chooses node n, then the pair consists of $benv(n)$ and the procedure corresponding to $proc(n)$. Note that the call graph guides the choice of the target since all paths in D correspond to call graph paths, and the bounded environment, which is computed by unfolding the k-bounded ancestors of the target. Importantly, the chosen node must be in A; this choice can be heuristic as long as no node in A is starved.

Summary Inference. Our algorithm learns new summaries for target predicates by applying four proof rules. For ease of exposition, we first describe these proof rules without induction (next subsection), followed by rules for induction and EC lemma. While these proof rules resemble those in SMASH [31], our queries involve k-bounded environments with $k \geq 1$ and our summaries are first-order theory formulas; in SMASH, queries use bounded environments with $k = 1$ and summaries are pre-/post-condition pairs over predicate abstractions. Additional proof rules specify the removal and addition of nodes in D and A. Appendix Aprovides the complete set of rules, omitted here due to space constraints.

6.2 Proof Rules without Induction

The algorithm updates the current *Goal* whenever a proof rule can be applied. Note that we are building a proof tree from the bottom-up, so an application of a rule here involves matching the conclusion to the current *Goal*. We abbreviate some common premises with names as shown in Fig. 3. For a node $n \in A$, let p be its procedure and b be its bounded environment. Also let *body* be the renaming of the body of p. The distinct feature of our algorithm is that the proof rules use only bounded environments.

The SAFE and UNSAFE rules (Fig. 4) allow us to conclude the safety or find a counterexample of the original program P using over- or under-approximate

AVAIL $n \in A$ PROC $p = proc(n)$ BENV $b = benv(n)$ IDX $i = idx(n)$
BODY $body = body_p[in, out \mapsto fv(callee_{bctx(n)}(i)), loc_p \mapsto fresh(loc_p)]$
PROP $hyp = \forall in, out \in fv(body).p(in, out) \Rightarrow indProp$
IND $indProp = \forall vars(\psi) \setminus (in_p \cup out_p).\mathbb{I}$

Fig. 3: Abbreviated premises, where $fresh(\vec{x})$ returns a vector $\vec{x'}$ of fresh variables.

SAFE
$$\frac{O \text{ is a solution for } \mathcal{C}}{D, A, O, U, L, \mathcal{C} \vdash \bot}$$

UNSAFE
$$\frac{\exists body \Rightarrow \bot \in \mathcal{C}.\widehat{body}_U \not\Rightarrow \bot}{D, A, O, U, L, \mathcal{C} \vdash \top}$$

OVER-OVER (OO)
$$\frac{\widehat{body}_O \Rightarrow \mathbb{I} \quad \overset{\text{AVAIL}}{\mathbb{I} \Rightarrow \neg\widehat{b}_O} \quad \overset{\text{PROC}}{O' = O[p \mapsto O[p] \wedge \mathbb{I}]} \quad \overset{\text{BENV}}{\overset{\text{BODY}}{D, A, O', U, L, \mathcal{C} \vdash Res}}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

UNDER-UNDER (UU)
$$\frac{\overset{\text{AVAIL}}{\overset{\text{PROC}}{\overset{\text{BENV}}{\overset{\text{IDX}}{body \Rightarrow p(in, out) \in \mathcal{C}}}}}}{\pi = body[in, out \mapsto fv(callee_{bctx(n)}(i)), loc_C \mapsto fresh(loc_C)]} \quad \widehat{\pi}_U \wedge \widehat{b}_U \not\Rightarrow \bot \quad U' = U[p \mapsto U[p] \vee \exists loc_C.\pi] \quad D, A, O, U', L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

Fig. 4: Proof rules without induction.

summaries, respectively. In the latter case, the underapproximate summaries demonstrate that there is no solution for set of CHCs $\mathcal{C}$. If either rule is applicable to the proof goal, we have found sufficient summaries.

The OVER-OVER (OO) rule (Fig. 4) can be used to update a predicate p's over-approximate summary. If the conjunction of over-approximation of $body_p$ and the bounded environment is unsatisfiable, then we can find an interpolant $\mathbb{I}$ and use it to refine the map O for p. The UNDER-OVER (UO) rule (Appendix A)is similar, except it uses an under-approximation of the environment.

Example 2. Recall the example in Fig. 1a. Row 5 in Table 1 shows the over-approximate summary $y \bmod 2 \neq 0$ for procedure g obtained as a result of UO.

The UNDER-UNDER (UU) rule (Fig. 4) can be used to update predicate p's under-approximation. Let π be the body of a CHC whose head is $p(\vec{y})$, where variables $\vec{y}$ have been renamed to the variables that p is applied to in $callee_{bctx(n)}(i)$ and the rest of the variables have been renamed to fresh ones. If the conjunction of the under-approximations of π and b is satisfiable, then we can update p's under-approximate summary U with $\exists loc_C.\pi$.

If the environment were unbounded, then this check being satisfiable would actually indicate a *concrete* counterexample, since the context would be an unfolding of a query CHC and UNSAFE would hold, but since our environment is *bounded*, the context may not be an unfolding of the query CHC, since it may be

Algorithm 2 Procedure to learn from a node.

```
1: procedure ProcessNode(n, Goal)
2:     for C ∈ 𝒞 with C.head = proc(n)(in, out) do
3:         if OU(n, C, Goal) then UU(n, C, Goal)
4:     if no UU call above returned true then
5:         if ¬OO(n, Goal) then UO(n, Goal)
6:     if no UU nor OO call above returned true then AddNodes(n, Goal)
7:     updated ← any summaries were updated above
8:     Processed(n, updated, Goal)
9:     return Goal
```

missing some constraints. We can only conclude that there might be a counterexample that involves unfolding this application of p. We want to remember the part that goes through p so that we do not need to unfold it in the full context and thus add $\exists loc_p.\pi$ to $U[p]$. The OVER-UNDER (OU) rule (Appendix A) is the same as UU but over-approximates the bounded environment.

Example 3. Recall the example in Fig. 1a. Row 7 in Table 1 shows the under-approximate summary $y \bmod 2 = 0$ for procedure `f` obtained as a result of OU.

6.3 Ordering and Conditions for SMT Queries

The way in which proof rules are applied to process a node is shown in Alg. 2. In the pseudocode, OO, UO, OU, and UU refer to attempts to apply the corresponding rules (e.g., OO(n, *Goal*) tries to apply the OO rule with $n \in A$ as the AVAIL premise). Rules that update under-approximations (UU, OU) are applied per CHC with head $proc(n)$[5], whereas rules that update over-approximations (OO, UO) are applied to the disjunction of all such CHCs' bodies. They return *true* upon successful application (and update *Goal*), or *false* otherwise.

If we have neither found any counterexamples through the bounded environment (i.e., all UU attempts failed), nor eliminated the bounded verification subtask (i.e., the OO attempt failed), then we try to derive new facts by adding new available nodes for the callees of $proc(n)$. Procedure AddNodes adds these nodes while avoiding adding redundant nodes to D (more details in Appendix C).If any summary updates were made for $proc(n)$, then the procedure Processed (line 8) will add the bounded parents of n to A, so that new information can be propagated to the parents' summaries. It then removes n from A.

6.4 Proof Rules for Induction

For programs with unbounded recursion, the OO and UO rules (Fig. 4) are insufficient for proving safety; we therefore extend the rules with induction where the goal is to show that the paths in the approximated bounded environment are spurious. For ease of exposition, we first discuss an extension of OO that does not use EC lemmas and then discuss one that does. (Corresponding extensions for rule UO are similar and can be found in Appendix A.)

[5] In the implementation, multiple checks can be done together.

OVER-OVER-IND (OOI)

$$\frac{\begin{array}{cccccc}\text{AVAIL} & \text{PROC} & \text{BENV} & \text{BODY} & \text{IND} & \text{PROP}\end{array}}{\widehat{body}_O \wedge hyp \Rightarrow \mathbb{I} \qquad \mathbb{I} \Rightarrow \neg\widehat{b}_O \qquad O' = O[p \mapsto O[p] \wedge \mathbb{I}] \qquad D, A, O', U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

OVER-OVER-IND-LEMMAS (OOIL)

$$\frac{\begin{array}{cccccc}\text{AVAIL} & \text{PROC} & \text{BENV} & \text{BODY} & \text{IND} & \text{PROP}\end{array}}{\begin{array}{c} S = assumps(n, D) \qquad \widehat{body}_O \wedge Inst(L) \wedge Inst(S) \wedge hyp \Rightarrow \mathbb{I} \qquad \mathbb{I} \Rightarrow \neg\widehat{b}_O \\ L' = L \cup \{\forall vars(S), in, out. \bigwedge S \Rightarrow (p(in, out) \Rightarrow \mathbb{I})\} \qquad D, A, O, U, L', \mathcal{C} \vdash Res \end{array}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

Fig. 5: Proof rules for induction.

Without EC lemmas. The rule OVER-OVER-IND (OOI) in Fig. 5 is a replacement for OO that uses induction to find new over-approximate facts. The first five premises are the same as in rule OO. As before, we aim to learn a refinement $\mathbb{I}$ for the over-approximate summary of p, where $\mathbb{I} \Rightarrow \neg\widehat{b}_O$.

The base case is that $\mathbb{I}$ over-approximates p for all CHCs that do not have any applications of p in their body, i.e. for all $body \Rightarrow p(\vec{y}) \in \mathcal{C}$ where p does not occur in $body$, $body \Rightarrow \mathbb{I}$. For the inductive step, we consider such CHCs where $body$ contains calls to P. The inductive hypothesis, which is captured by formula hyp, is that $\mathbb{I}$ over-approximates all recursive calls to p inside these bodies. We check both the base case and the inductive step at once with the implication $\widehat{body}_O \wedge hyp \Rightarrow \mathbb{I}$. If the induction succeeds, then we strengthen $O[p]$ with $\mathbb{I}$.

With EC lemmas. The OVER-OVER-IND-LEMMAS (OOIL) proves weaker properties than OOIL by doing induction under certain *assumptions*. These properties are *EC lemmas*.

OOIL makes assumptions for current node n and performs induction using these assumptions and known EC lemmas. In particular, $assumps(n, D)$ is a set of assumptions $\{a_i \mid 1 \leq i \leq j\}$ for some $j \geq 0$. When $j = 0$, the set of assumptions is empty, and OOIL has the same effect as applying OOI. Each assumption a_i is of the following form:

$$q_i(in_{q_i}, out_{q_i}) \Rightarrow \forall vars(b_i) \setminus in_{q_i} \cup out_{q_i}.\neg b_i,$$

where q_i is the predicate for an ancestor of n and q_i is called by target p in some CHC. The ancestor node's bounded environment is b_i. Intuitively, each assumption is that the ancestor's bounded verification subtask has been discharged.

The *Inst* function takes a set of formulas, conjoins them, and replaces each application of an uninterpreted predicate with its interpretation in O. When applied to a set of assumptions S, it has an additional step that precedes the others: it first adds a conjunct $a_i[in_{q_i} \mapsto x, out_{q_i} \mapsto y]$ for each predicate application $q_i(x, y)$ in $body$ to each each element $a_i \in S$. This corresponds to applying the assumption in the induction hypothesis. If induction succeeds, we learn the EC lemma that $\mathbb{I}$ over-approximates $p(in, out)$ under the assumptions S.

Example 4. In §2, when we chose procedure `e` and proved an EC lemma, we used $j = 1$ to make an assumption about its caller `o`.

Appendix A contains additional rules that allow lemmas to be simplified. There may be multiple attempts at applying the OOIL proof rule with different j values. For scalability, we require that j not exceed the bound k used for bounded contexts, limiting the number of these attempts.

6.5 Correctness and Progress

The correctness and progress claims for Alg. 1 are stated below.

Theorem 1 (Correctness). *Alg. 1 returns safe (resp. unsafe) only if the program with entry point* `main` *never violates the assertion (resp. a path in* `main` *violates the assertion).*

Proof. (Sketch) The CHC encoding is such that there is solution to the system of CHCs $\mathcal{C}$ iff the program does not violate the assertion. As a result, if the over-approximate summaries O constitute a solution and proof rule SAFE can be applied, the program does not violate the assertion. The under-approximate summaries U in every proof subgoal are guaranteed to be such that for any $p \in \mathcal{C}$, $U[p]$ implies any over-approximation $O[p]$. If UNSAFE can be applied, then the under-approximate summaries U imply that there is no possible solution O. The summaries in U can be used to reconstruct a counterexample path through the original program in this case.

Theorem 2 (Progress). *Processing a node in the derivation tree leads to at least one new (non-redundant) query.*

Proof. (Sketch) Initially, no nodes in A have been processed, and after a node is processed, it is removed from the derivation tree. The only way that a node can be processed and not have a new query made about it is if an already-processed node is re-added to A and this node does not have a new query that can be made about it. The AddNodes and MakeUnavailable procedures are the only ones that add nodes to A. The AddNodes procedure, by definition, will only add a node to A if there is a new query that can be made about it. MakeUnavailable only adds bounded parents of nodes whose summaries were updated. For any such bounded parent, at least one approximation of its procedure's body must be different than it was the last time the bounded parent was processed, since one of its callee's summaries was updated.

Limitations. If the underlying solver is unable to find appropriate interpolants, the algorithm may generate new queries indefinitely. (The underlying problem is undecidable, so this is not unusual for modular verifiers.) Note, however, that because environments are bounded, each query's size is restricted.

7 Evaluation and Results

We implemented our algorithm in a tool called Clover on top of CHC solver FreqHorn [25] and SMT solver Z3 [50]. We evaluated Clover and compared it with existing CHC-based tools on three sets of benchmarks (described later) that comprise standard collections and some new examples that include mutual recursion. We aimed to answer the following questions in our evaluation:

- Is Clover able to solve standard benchmarks?
- Is Clover more effective than other tools at handling mutual recursion?
- To what extent do EC lemmas help Clover solve benchmarks?
- How does the bound k for environments affect Clover's performance?

We compared Clover against tools entered in the annual CHC-solver competition (CHC-Comp) in 2019: Spacer [42], based on PDR [12]; Eldarica [39], based on CEGAR [17]; HoIce [13], based on ICE [29]; PCSat [55]; and Ultimate Unihorn [22], based on trace abstraction [35].

For all experiments, we used a timeout of 10 minutes (as used in CHC-Comp). We ran Clover on a MacBook Pro, with a 2.7GHz Intel Core i5 processor and 8GB RAM, but the other tools were run using StarExec [58]. Clover was not run on StarExec due to difficulties with setting up the tool with StarExec[6].

7.1 Description of Benchmarks

To evaluate Clover, we gathered three sets of varied benchmarks. The first set's benchmarks range from ˜10-7200 lines, and the latter two sets have smaller but nontrivial code (˜100 lines). The latter two sets were manually encoded into CHCs, and we plan to contribute them to CHC-Comp. Additional details follow.

CHC-Comp. We selected 101 benchmarks from CHC-Comp [14] that were contributed by HoIce and PCSat, since their encodings preserve procedure calls and feature nonlinear CHCs (which can represent procedures with multiple callees per control-flow path)[7].

Real-World. Two families of benchmarks are based on real-world code whose correctness has security implications. The *Montgomery* benchmarks involve properties about the sum of Montgomery representations [41] of concrete numbers. The *s2n* benchmarks are based on Amazon Web Services' s2n library [3] and involve arrays of unbounded length (not handled by the tool PCSat).

Mutual Recursion. This set of benchmarks containing mutual recursion was created because few CHC-Comp benchmarks exhibit mutual recursion, likely due to lack of tool support. *Even-Odd* benchmarks involve various properties of `e` and `o` (defined as in §2) and extensions that handle negative inputs. Another benchmark family is based on the *Hofstadter* Figure-Figure sequence [38]. *Mod*

[6] We expect that our platform is less performant than the StarExec platform
[7] We did not compare against FreqHorn since it cannot handle nonlinear CHCs.

Table 2: Number of Benchmarks Solved by Clover and Competing Tools.

	Clover				Spacer	Elda-rica	HoIce	PCSat	Ulti-mate Auto-mizer
	k=2	k=9	k=10	k=10, no EC lem-mas					
CHC-Comp (101)	80	77	77	72	93	94	92	81	76
Montgomery (12)	0	11	12	12	5	12	12	3	11
s2n (4)	3	4	4	4	3	0	2	N/A	4
Even-Odd (24)	24	24	24	0	12	0	9	0	0
Hofstadter (5)	4	5	4	5	1	4	5	5	0
Mod n (15)	0	15	15	0	0	0	0	0	0
Combination (2)	0	2	2	0	0	0	0	0	0
Total Solved (163)	111	138	138	93	114	110	120	89	91

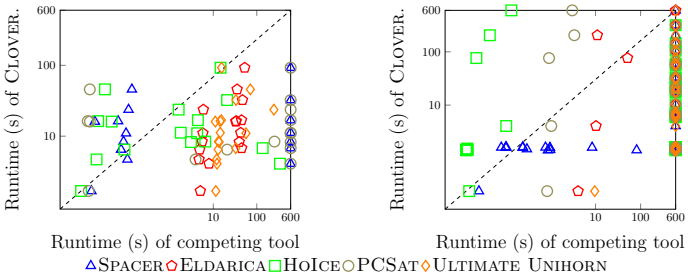


Fig. 6: Timing results for the Real World (left) and Mutual Recursion (right) benchmarks. Points below the diagonal line are those for which Clover outperforms the corresponding tool. Points on the right edge indicate timeouts of the other tool.

n benchmarks consider mutually-recursive encodings of $\lambda x.x \bmod n = 0$ for $n = 3, 4, 5$. These serve as proxies for recursive descent parsers, which may have deep instances of mutual recursion. We could not directly conduct experiments on such parsers, since existing front-ends [33, 21] cannot handle them. *Combination* benchmarks result from combining Montgomery and Even-Odd benchmarks.

7.2 Results and Discussion

Table 2 gives a summary of results. It reports the number of benchmarks solved for each benchmark set by Clover with bound parameter k being 2, 9, and 10 (the best-performing bounds for the three benchmark sets) and by the other tools. It also reports results for Clover with $k = 10$ but without EC lemmas. Fig. 6 show the timing results for other tools against Clover for Real-World and Mutual Recursion benchmarks.

Efficacy on standard benchmarks. As can be seen in Table 2, Clover performs comparably with other tools on the CHC-Comp benchmarks, and significantly outperforms them on the other two sets of benchmarks. We expect that we can further improve the performance of Clover with additional optimizations and heuristics, such as those that improve the quality of interpolants.

Efficacy on Mutual Recursion benchmarks. Table 2 and Fig. 6 demonstrate that Clover is more effective and often more efficient at solving Mutual Recursion benchmarks than the other tools. Few tools are able to handle

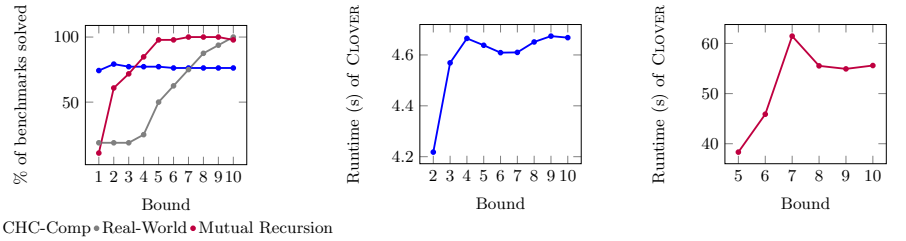


Fig. 7: Left: Percentage of benchmarks Clover solves with different bounds on different benchmark categories; **Center**, **Right**: Timing results on a representative benchmark from CHC-Comp and Mutual Recursion, respectively.

the Even-Odd benchmarks, which Clover (with EC lemmas) can solve at any bound value greater than 2. Other tools are unable to solve even half of the Mutual Recursion benchmarks, reinforcing that Clover is a useful addition to existing tools that enables handling of mutual recursion as a first class concern.

Usefulness of EC lemmas. Running Clover with and without EC lemmas using bound $k = 10$ revealed their usefulness for many of the benchmarks. In particular, the columns for bound 10 with and without EC lemmas in Table 2 show that EC lemmas are needed to allow Clover to solve several CHC-Comp benchmarks and all the Mutual Recursion benchmarks except the Hofstadter ones. These results indicate that Clover's ability to outperform other tools on the these benchmarks relies on EC lemmas.

Comparison of Different Bounds. Fig. 7 (left) shows the number of benchmarks successfully solved by Clover in each set as the bound value is varied. Running Clover with *too small* a bound impedes its ability to prove the property or find a counterexample, since the environment is unable to capture sufficient information.On the other hand, running Clover with *too large* a bound affects the runtime negatively. This effect can be observed in Fig. 7 center and right, which show how the runtime varies with the bound for a representative benchmark from the CHC-Comp and Mutual Recursion sets, respectively. Note that at a bound $k < 2$, Clover does not solve the given CHC-Comp benchmark, and at $k < 5$, Clover does not solve the given Mutual Recursion benchmark. These results confirm the expected trade-off between scalability and environment relevance. The appropriate trade-off – i.e., the best bound parameter to use – depends on the type of program and property. As seen in Fig. 7 (left), the bound values that lead to the most benchmarks being solved differ per benchmark set. Rather than having a fixed bound, or no bound at all, the ability to choose the bound parameter in Clover allows the best trade-off for a particular set of programs. If the best bound is not known a priori, bound parameters of increasing size can be determined empirically on representative programs.

We also report data on how the number and solving time for each type of SMT query varies with the bound k, averaged over benchmarks in each set. Fig. 8 shows the statistics on the average number of queries of each type (top), on the average time taken to solve the query (bottom). These data are from all runs for which Clover is successful and gives an answer of *safe* or *unsafe*.

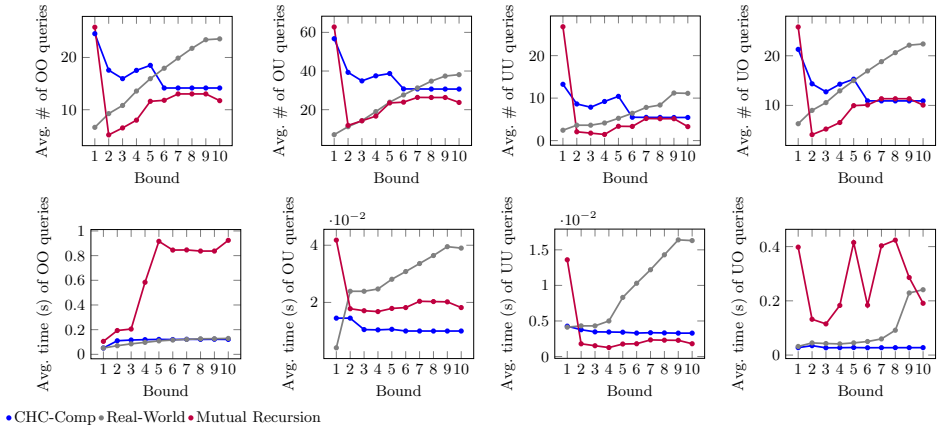


Fig. 8: Average statistics (**top four** plots: number, **bottom four**: solve times) of SMT queries made by Clover as the bound changes (for successful runs).

We can use these data along with the data in Fig. 7 to (roughly) compare an approach restricted to $k = 1$ with an approach that allows $k > 1$ in bounded environments. Note that Clover differs significantly in other respects from tools like Spacer and Smash that enforce $k = 1$ in environments[8], making it difficult to perform controlled experiments to compare this aspect alone.

Note from Fig. 8 that for the CHC-Comp and Mutual Recursion sets of benchmarks, the number of SMT queries of all types is lower at $k > 1$ in comparison to $k = 1$. This result indicates that benchmarks that can be solved with $k > 1$ require on average fewer updates to procedure summaries than are needed on average for benchmarks that can be solved with $k = 1$, confirming the benefit of improved relevance when going beyond a restricted environment with $k = 1$. The data for the Real-World does not follow this trend because a higher bound ($k = 10$) is needed to solve the examples (as can be seen in Fig. 7).

From Fig. 8, it is clear that the OU and UU queries are cheaper than OO and UO queries, which is expected since the latter require over-approximating the target's body. Unsurprisingly, OO queries are the most expensive. Average times of non-OO queries for $k > 1$ are lower than (or about the same as) average times for $k = 1$ for the CHC-Comp and Mutual Recursion sets but continue to increase with k in the Real-World set because solving the Montgomery benchmarks relies on propagating under-approximations from increasingly large call graph depths.

8 Related Work

There is a large body of existing work that is related in terms of CHC solving, program analysis, and specification inference.

[8] Unlike Spacer it does not use PDR to derive invariants, and unlike Smash it is not limited to predicate abstractions.

8.1 CHC-solving for Program Verification

Program verification problems, including modular verification, can be encoded into systems of CHCs [32, 49, 33, 21]. There are many existing CHC-solver based tools [13, 42, 48, 59, 26, 15, 32, 39, 61] that can solve such systems. Clover has many algorithmic differences from these efforts.

Most existing tools do not place any bounds on the environments (if they are used at all). This includes approaches that unfold a relation at each step [48, 59] and CEGAR-based approaches [32, 39] where counterexamples can be viewed as environments. These tools face scalability issues as environments grow; Duality makes larger interpolation queries as more relations are unfolded [49], and Eldarica makes larger tree/disjunctive interpolation queries for counterexamples that involve more procedures [39].

Spacer [42], which is based on PDR [12, 23], considers bounded environments but only allows a bound of one ($k = 1$). The difference between Duality and a PDR-like approach has been referred to as the *variable elimination trade-off* [48], where eliminating too many variables can lead to over-specialization of learned facts (PDR) and eliminating no variables can lead to larger subgoals (Duality). Our parameterizable bounded environments enable a trade-off between the two. Another significant difference between Spacer and Clover is that the former uses PDR-style bounded assertion maps to perform induction, whereas we use induction explicitly and derive EC lemmas. Duality may also implicitly use assumptions, and some other tools [13, 59] learn lemmas with implications, but none of them learn lemmas in the form of EC lemmas.

HoIce [13], FreqHorn [26], and LinearArbitrary [61] are based on guessing summaries and do not have any notion of environments similar to ours. All of these approaches have trade-offs between scalability of the search space and expressivity of guessed summaries.

8.2 Program Analysis and Verification

Techniques such as abstract interpretation [18, 19, 24] and interprocedural dataflow analysis [56, 54] can infer procedure summaries and perform modular verification. These approaches often use fixed abstractions and path-insensitive reasoning, which may result in over-approximations that are too coarse for verification.

The software model checker Bebop [6] in SLAM [7] extended interprocedural dataflow analysis with path sensitivity. Related model checkers include a direct precursor to Duality [47] and other adaptations of PDR to software [37, 16]. Of these, GPDR [37] is similar to Spacer, but lacks modular reasoning and under-approximations. Specification inference (including Houdini-style learning [28]) has also been used to enable modular verification of relational programs [43, 52].

Another tool Smash [31] is closely related to our work. It uses over- and under-approximate procedure summaries, and alternation between them. However, it does not have any notion of a parameterizable bounded environment. The environment for a procedure call is expressed as a pair of a precondition and a postcondition, where the former is an under-approximation of the program execution preceding the call, and the latter is an over-approximation of

the program execution following the call. These environments are thus bounded environments with a fixed bound of 1. More importantly, procedure summaries in Smash are comprised of predicate abstractions. In contrast, our summaries are richer formulas in first-order logic theories. We do not rely on predicate abstraction unlike Smash and other related tools [31, 30, 34].

8.3 Specification Inference

Existing work on specification inference is also relevant. Many past efforts [2, 4, 9, 53, 57, 60] focused on learning coarse interface specifications or rules specifying the correct usage of library API calls, rather than learning logical approximations of procedures. Other specification inference techniques learn procedure summaries for library API procedures by using abstract interpretation [36, 19] or learn information-flow properties about the procedures [45, 52]. Other related work [1] infers maximal specifications for procedures with unavailable bodies, and other techniques assume an angelic environment setting [11, 20] – specifications inferred by these techniques may not be valid over-approximations. Another technique [5] also uses interpolation to infer over-approximate summaries but is not applicable to recursive programs.

9 Conclusions

We have presented a modular algorithm for generating procedure summaries and safety verification of interprocedural programs that addresses the challenges of handling mutual recursion and scalability of SMT queries. The novel features of our algorithm are use of bounded environments to limit the size of SMT queries, and a mechanism for performing induction under assumptions that uses these bounded environments to learn EC lemmas that capture relationships between summaries of procedures on call paths in the program.

We have implemented our algorithm in a CHC-based tool called Clover. An evaluation demonstrates that Clover is competitive with state-of-the-art tools on benchmarks from CHC-Comp and based on real-world examples, and is especially effective at solving benchmarks containing mutual recursion. Our algorithm can also be combined with existing invariant-generation techniques to successfully solve benchmarks with unbounded arrays.

Acknowledgements This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656466. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was supported in part by the National Science Foundation award FMitF 1837030.

References

1. Albarghouthi, A., Dillig, I., Gurfinkel, A.: Maximal specification synthesis. In: POPL. pp. 789–801. ACM (2016)

2. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL. pp. 98–109. ACM (2005)
3. Amazon Web Services: https://github.com/awslabs/s2n (2019)
4. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL. pp. 4–16. ACM (2002)
5. Asadi, S., Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Even-Mendoza, K., Sharygina, N., Chockler, H.: Function summarization modulo theories. In: LPAR. EPiC Series in Computing, vol. 57, pp. 56–75. EasyChair (2018)
6. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. In: PASTE. pp. 97–103. ACM (2001)
7. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. Lecture Notes in Computer Science, vol. 2102, pp. 260–264. Springer (2001)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885. IOS Press (2009)
9. Beckman, N.E., Nori, A.V.: Probabilistic, modular and scalable inference of typestate specifications. In: PLDI. pp. 211–221. ACM (2011)
10. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: LPAR (short papers). EPiC Series in Computing, vol. 35, pp. 15–27. EasyChair (2015)
11. Blackshear, S., Lahiri, S.K.: Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In: PLDI. pp. 209–218. ACM (2013)
12. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI. Lecture Notes in Computer Science, vol. 6538, pp. 70–87. Springer (2011)
13. Champion, A., Kobayashi, N., Sato, R.: Hoice: An ice-based non-linear horn clause solver. In: APLAS. Lecture Notes in Computer Science, vol. 11275, pp. 146–156. Springer (2018)
14. CHC-Comp: https://chc-comp.github.io (2019)
15. Chen, Y., Hsieh, C., Tsai, M., Wang, B., Wang, F.: Verifying recursive programs using intraprocedural analyzers. In: SAS. Lecture Notes in Computer Science, vol. 8723, pp. 118–133. Springer (2014)
16. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV. Lecture Notes in Computer Science, vol. 7358, pp. 277–293. Springer (2012)
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)
18. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Recursive Procedures. In: IFIP. pp. 237–278 (1977)
19. Cousot, P., Cousot, R., Fähndrich, M., Logozzo, F.: Automatic inference of necessary preconditions. In: Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI. pp. 128–148 (2013)
20. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: Precise verification modulo unknowns. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 324–342. Springer (2015)
21. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verimap: A tool for verifying programs through transformations. In: TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 568–574. Springer (2014)
22. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer. In: HCVS/PERR. EPTCS, vol. 296, pp. 42–47 (2019)
23. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134. FMCAD Inc. (2011)

24. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: FoVeOOS. Lecture Notes in Computer Science, vol. 6528, pp. 10–30. Springer (2010)
25. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: FMCAD. pp. 100–107. IEEE (2017)
26. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving Constrained Horn Clauses Using Syntax and Data. In: FMCAD. pp. 170–178. ACM (2018)
27. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified Invariants via Syntax-Guided Synthesis. In: CAV, Part 1. Lecture Notes in Computer Science, vol. 11561, pp. 259–277. Springer (2019)
28. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for esc/java. In: FME. Lecture Notes in Computer Science, vol. 2021, pp. 500–517. Springer (2001)
29. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 69–87 (2014)
30. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: CONCUR. Lecture Notes in Computer Science, vol. 2154, pp. 426–440. Springer (2001)
31. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: POPL. pp. 43–56. ACM (2010)
32. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. pp. 405–416. ACM (2012)
33. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: CAV (1). Lecture Notes in Computer Science, vol. 9206, pp. 343–361. Springer (2015)
34. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: A software model-checker for verification and refutation. In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 170–174. Springer (2006)
35. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: SAS. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009)
36. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE. pp. 31–40. ACM (2005)
37. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT. Lecture Notes in Computer Science, vol. 7317, pp. 157–171. Springer (2012)
38. Hofstadter, D.R., et al.: Gödel, Escher, Bach: an eternal golden braid, vol. 20. Basic books New York (1979)
39. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: FMCAD. pp. 1–7. IEEE (2018)
40. Ivancic, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Tokuoka, H., Imoto, T., Miyazaki, Y.: DC2: A framework for scalable, scope-bounded software verification. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 133–142 (2011)
41. Koc, C.K., Acar, T., Kaliski, B.S.: Analyzing and comparing montgomery multiplication algorithms. IEEE micro **16**(3), 26–33 (1996)
42. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016)
43. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: ESEC/SIGSOFT FSE. pp. 345–355. ACM (2013)

44. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Proceedings of the International Conference on Computer Aided Verification (CAV). pp. 427–443 (2012)
45. Livshits, V.B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: specification inference for explicit information flow problems. In: PLDI. pp. 75–86. ACM (2009)
46. McMillan, K.L.: Interpolation and sat-based model checking. In: CAV. Lecture Notes in Computer Science, vol. 2725, pp. 1–13. Springer (2003)
47. McMillan, K.L.: Lazy annotation for program testing and verification. In: CAV. Lecture Notes in Computer Science, vol. 6174, pp. 104–118. Springer (2010)
48. McMillan, K.L.: Lazy annotation revisited. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 243–259. Springer (2014)
49. McMillan, K.L., Rybalchenko, A.: Solving constrained horn clauses using interpolation (2013)
50. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
51. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
52. Pick, L., Fedyukovich, G., Gupta, A.: Automating modular verification of secure information flow. In: FMCAD. pp. 158–168. TU Wien Academic Press, IEEE (2020)
53. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: PLDI. pp. 123–134. ACM (2007)
54. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. pp. 49–61. ACM Press (1995)
55. Satake, Y., Kashifuku, T., Unno, H.: PCSat: Predicate constraint satisfaction (2019), https://chc-comp.github.io/2019/chc-comp19.pdf
56. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. pp. 189–233 (1981)
57. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA. pp. 174–184. ACM (2007)
58. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: a cross-community infrastructure for logic solving. In: International joint conference on automated reasoning. pp. 367–373. Springer (2014)
59. Unno, H., Torii, S., Sakamoto, H.: Automating induction for solving horn clauses. In: CAV (2). Lecture Notes in Computer Science, vol. 10427, pp. 571–591. Springer (2017)
60. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: ICSE. pp. 282–291. ACM (2006)
61. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: PLDI. pp. 707–721. ACM (2018)

A Full Set of Derivation Rules

AVAIL	$n \in A$	PROC	$p = proc(n)$
BENV	$b = benv(n)$	IDX	$i = idx(n)$
NODE	$n \in D.N$		
PATH	$C = body \Rightarrow p(in, out) \in \mathcal{C} \wedge$		
	$\pi = body[in, out \mapsto fv(callee_{bctx(n)}(i)), local_C \mapsto fresh(local_C)]$		
BODY	$body = body_p[in, out \mapsto fv(callee_{bctx(n)}(i)), loc_p \mapsto fresh(loc_p)]$		
CALL	$callee_C(idx(n')) = proc(n')(\vec{x})$		
NCTX	$ctx(n')$ is the unfolding of $proc(n')(\vec{x})$ in C		
PROP	$hyp = \forall in, out \in vars(body).p(in, out) \Rightarrow indProp$		
IND	$indProp = \forall vars(\psi) \setminus (in_p \cup out_p).\mathbb{I}$		

SAFE

$$\frac{O \text{ is a solution for } \mathcal{C}}{D, A, O, U, L, P \vdash \bot}$$

UNSAFE

$$\frac{\exists body \Rightarrow \bot \in \mathcal{C}.\widehat{body}_U \not\Rightarrow \bot}{D, A, O, U, L, P \vdash \top}$$

OVER-OVER (OO)

$$\frac{\begin{array}{cccc} AVAIL & PROC & BENV & BODY \\ \widehat{body}_O \Rightarrow \mathbb{I} & \mathbb{I} \Rightarrow \neg\widehat{b}_O & O' = O[p \mapsto O[p] \wedge \mathbb{I}] & D, A, O', U, L, \mathcal{C} \vdash Res \end{array}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

UNDER-OVER (UO)

$$\frac{\begin{array}{cccc} AVAIL & PROC & BENV & BODY \\ & \widehat{body}_O \Rightarrow \mathbb{I} & \mathbb{I} \Rightarrow \neg\widehat{b}_U & \\ O' = O[p \mapsto O[p] \wedge \mathbb{I}] & & D, A, O', U, L, P \vdash Res & \end{array}}{D, A, O, U, L, P \vdash Res}$$

UNDER-UNDER (UU)

$$\frac{\begin{array}{ccccc} AVAIL & PROC & BENV & IDX & PATH \\ \widehat{\pi}_U \wedge \widehat{b}_U \not\Rightarrow \bot & U' = U[p \mapsto U[p] \vee \exists loc_C.\pi] & & D, A, O, U', L, \mathcal{C} \vdash Res & \end{array}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

OVER-UNDER (OU)

$$\frac{\begin{array}{ccccc} AVAIL & PROC & BENV & IDX & PATH \\ \widehat{\pi}_U \wedge \widehat{b}_O \not\Rightarrow \bot & U' = U[p \mapsto U[p] \vee \exists loc_C.\pi] & & D, A, O, U', L, \mathcal{C} \vdash Res & \end{array}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

ADD-NODE (AN)

$$\frac{\begin{array}{ccccc} AVAIL & PROC & PATH & CALL & NCTX \\ D'.E = D.E \cup \{n \to n'\} & & \forall n'' \in D.N.bctx(n') \neq bctx(n'') & & \\ A' = A \cup \{n'\} & & D', A', O, U, L, \mathcal{C} \vdash Res & & \end{array}}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

MAKE-AVAILABLE (MA)

$$\frac{NODE \quad PROC \quad PATH \quad CALL \quad NCTX \qquad n'' \in D.N \qquad bctx = bctx(n'') \qquad A' = A \cup \{n''\} \qquad D, A', O, U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

MAKE-UNAVAILABLE (MU)

$$\frac{AVAIL \qquad D, (A \setminus \{n\}), O, U, L, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

OVER-OVER-IND-LEMMAS (OOIL)

$$\frac{AVAIL \quad PROC \quad BENV \quad BODY \quad IND \quad PROP \qquad S = assumps(n, D) \qquad \widehat{body}_O \wedge Inst(L) \wedge Inst(S) \wedge hyp \Rightarrow \mathbb{I} \qquad \mathbb{I} \Rightarrow \neg\widehat{b}_O \qquad L' = L \cup \{\forall vars(S), in, out. \bigwedge S \Rightarrow (p(in, out) \Rightarrow \mathbb{I})\} \qquad D, A, O, U, L', \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

UNDER-OVER-IND-LEMMAS (UOIL)

$$\frac{AVAIL \quad PROC \quad BENV \quad BODY \quad IND \quad PROP \qquad S = underAssumps(n, D) \qquad \widehat{body}_O \wedge Inst(L) \wedge Inst(S) \wedge hyp \Rightarrow \mathbb{I} \qquad \mathbb{I} \Rightarrow \neg\widehat{b}_U \qquad L' = L \cup \{\forall vars(S), in, out. \bigwedge S \Rightarrow (p(in, out) \Rightarrow \mathbb{I})\} \qquad D, A, O, U, L', \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

REDUCE-LEMMAS (RL)

$$\frac{ec = \forall vars(S), in, out. \bigwedge S \Rightarrow \big(p(in, out) \Rightarrow \psi\big) \in L \qquad a \in S \qquad p' \in P \qquad (p'(in, out) \Rightarrow O[p']) \Rightarrow a \qquad S' = S \setminus \{a\} \qquad L' = (L \setminus \{ec\}) \cup \big\{\forall vars(S'), in, out. \bigwedge S' \Rightarrow \big(p(in, out) \Rightarrow \psi\big)\big\} \qquad D, A, O, U, L', \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

ELIM-LEMMAS (EL)

$$\frac{ec = \forall in, out. \top \Rightarrow (p(in, out) \Rightarrow prop) \in L \qquad O' = O[p \mapsto O[p] \wedge prop] \qquad D, A, O', U, L \setminus \{ec\}, \mathcal{C} \vdash Res}{D, A, O, U, L, \mathcal{C} \vdash Res}$$

B Heuristics

B.1 Prioritizing choice of node

The VERIFY procedure from Fig. 1 employs a heuristic to choose which node in the set A to call PROCESSNODE on next. The factors that contribute toward an node's priority are as follows, with ties in one factor being broken by the

next factor, where $\mathit{depth}(n)$ denotes the depth of node n in D and $\mathit{previous}(n)$ denotes the number of times that the node n has been chosen previously:

- A lower $\alpha * \mathit{depth}(n) + \beta * \mathit{previous}(n)$ score gives higher priority, where α and β are weights
- A lower call graph depth of $\mathit{proc}(n)$ gives higher priority
- A later index $\mathit{callee}^{-1}_{ctx(n)}$ gives higher priority

We prioritize nodes n with lower $\mathit{depth}(n)$ values because they are more likely to help propagate learned summaries up to the *main* procedure's callees. This priority is moderated by the $\mathit{previous}(n)$ score which should prevent the starvation of nodes with larger $\mathit{depth}(n)$ values. Our current heuristic search is more BFS-like, but for some examples, a DFS-like search is better. We plan to improve our heuristics in future work.

B.2 Avoiding Redundant Queries

If we have previously considered a node n that we are now processing, we can avoid making the same queries that we have previously made. E.g., if none of the over-approximate summaries for any of the predicates in $\mathit{benv}(n)$ nor any of over-approximate summaries for any of the procedures called by $\mathit{proc}(n)$ have been updated since the last time n was processed, we do not need to redo the over-over check.

B.3 Learning Over-approximate Bodies

Although there are many existing methods to interpolate, in many cases they are useless (recall our motivating example where an interpolant is just $\top$). To improve our chances of learning a refinement for an over-approximate summary, whenever we apply one of the proof rules that involves over-approximating the procedure body (e.g., OO, UO, OOIL, UOIL), we ensure that we at least learn the result of over-approximating the procedure body as an over-approximate fact. For example, if we consider doing this for OO, we would simply replace premise $O' = O[p \mapsto O[p] \wedge \mathbb{I}]$ with $O' = O[p \mapsto O[p] \wedge \mathbb{I} \wedge \exists \mathit{loc}_p.\widehat{\mathit{body}}_O]$. Note that the result of applying quantifier elimination to $\mathbb{I} \wedge \exists \mathit{loc}_p.\widehat{\mathit{body}}_O$ is also an interpolant. Similarly, if we consider doing this for OOIL, we replace the goal $D, O, U, L', P \vdash \mathit{Res}$ with $D, O[p \mapsto \exists \mathit{loc}_p.\widehat{\mathit{body}}_O], U, L', P \vdash \mathit{Res}$.

B.4 Preventing summaries from growing too large

Although we want to increase our chances of learning useful refinements of over-approximations as we have just discussed, we still wish to prevent summaries from becoming too complicated. We can achieve this in a few ways.

Algorithm 3 Procedure for adding nodes in derivation tree

```
procedure AddNodes(n, Goal)
    for CHC C ∈ 𝒞 where C.head is an application of proc(n) do
        for p(x⃗_i) = callee_C(i) that is not a callee of ctx(n') where n' → n ∈ D.E do
            if ¬TryAddNode(n, C.body, p, i, Goal) then
                MakeAvailableIfNew(n, C.body, p, i, Goal)
```

Quantifier Elimination One way that we can achieve this is to use quantifier elimination or an approximation thereof on each conjunct (resp. disjunct) that we add to an over- (resp. under-) approximate summary. For example, we can replace $U' = U[p \mapsto U[p] \vee \exists loc_C.\pi]$ with $U' = U[p \mapsto U[p] \vee \text{QE}(\exists loc_C.\pi)]$ in the UU rule. We illustrate how to do this using two examples:

- Instead of using premise $O' = O[p \mapsto O[p] \wedge \mathbb{I} \wedge \exists loc_p.\widehat{body_O}]$ for the OO rule as just discussed, we use the following premise: $O' = O[p \mapsto O[p] \wedge \mathbb{I} \wedge \text{QE}(\exists loc_p.\widehat{body_O})]$
- We can also apply this to properties we learn by induction. Instead of using the premise $L' = L \cup \{\forall vars(A). \bigwedge A \Rightarrow (p(in, out) \Rightarrow indProp)$ for rule OOIL, use the following premise: $L' = L \cup \{\forall vars(A). \bigwedge A \Rightarrow (p(in, out) \Rightarrow QE(indProp))$
- Replace premise $U' = U[p \mapsto U[p] \vee \exists loc_C.\pi]$ with $U' = U[p \mapsto U[p] \vee \text{QE}(\exists loc_C.\pi)]$ in the UU rule.

This use of QE leads to quantifier-free summaries.

When we update over- (resp. under-) approximate summaries, we can use over- (resp. under-) approximate QE. By comparison, under- (resp. over-) approximate QE would lead to unsoundness. Approximating QE is not only cheaper but can also further simplify the resulting summary.

Selective Updates We can also prevent summaries from growing too quickly syntactically by only performing semantic updates. For example, consider O from the goal of the OO rule and O' from its subgoal. If $O[p] \Rightarrow O'[p]$, then although $O'[p]$ contains more conjuncts than $O[p]$, it does not provide any new information. In this case, we avoid the update and simply use O in the subgoal instead of O'. Similarly, if we consider U from the goal of UU and U' from its subgoal, then we only want to update the under-approximation if we have that $U'[p] \not\Rightarrow U[p]$. Over-approximate summaries become monotonically more constrained, so if $O[p] \Rightarrow O'[p]$, then $O[p] \Leftrightarrow O'[p]$ must hold. Under-approximations become monotonically less constrained.

C Addition of Nodes in Derivation Tree

The AddNodes procedure is shown in Alg. 3. For every CHC C with an application of $p = proc(n)$ as its head, it calls procedure TryAddNode$(n, C.body, p, i, Goal)$, which tries to apply AN to $Goal$ with premises $n \in A$ (AVAIL), CHC C whose body can be renamed to get π (PATH), and call to

$p = \mathit{proc}(n') \in P$ at index i in $C.\mathit{body}$ (CALL). If TryAddNode succeeds in applying AN, then it updates *Goal* to be the subgoal of the application and returns *true*. If it fails, then it performs no updates and returns *false*. If TryAddNode fails, then there is already a node n'' in D with the same bounded environment that the new node n' would have. In this case, AddNodes calls MakeAvailableIfNew, which applies MA if either of the following hold:

- n'' has never been processed before
- n'' has previously been processed with summaries O_{prev} and U_{prev} and the body *body* of $\mathit{proc}(n)$ or the bounded environment *benv* for n'' has a different over- or under-approximation than before, i.e., $\widehat{\mathit{body}}_{M_{prev}} \neq \widehat{\mathit{body}}_M$ or else $\widehat{\mathit{benv}}_{M_{prev}} \neq \widehat{\mathit{benv}}_M$ for $M \in \{O, U\}$

Similarly to TryAddNode, the procedure MakeAvailableIfNew$(n, C.\mathit{body}, p, i, \mathit{Goal})$ applies MA with premises $n \in D.N$ (NODE), CHC C whose body can be renamed to get π (PATH), and callee $\mathit{proc}(n') \in P$ at index i in C (CALL). Both TryAddNode and MakeAvailableIfNew have the side-effect of updating *Goal* to be the subgoal of the applied rule (if any).