Learning Summaries of Recursive Functions

Yu-Fang Chen Bow-Yaw Wang Kai-Chun Yang
Institute of Information Science
Academia Sinica, Taiwan

***Abstract*—We describe a learning-based approach for verifying recursive functions. The Boolean formula learning algorithm CDNF is used to automatically infer function summaries for recursive functions. In contrast to traditional iterative fixpoint computation-based approaches, ours can quickly guess summaries and verify purported summaries. When purported summaries are incorrect, the learning algorithm refines them by posing queries. We solve examples that are unattainable by a mature model checker for recursive programs.**

I. Introduction

A *function summary* is a formula encoding an over-approximation of function behaviors. A summary can be viewed as a well-specified interface of a function. Computing function summaries is an important technique in program verification. One can replace a function with its summary when he tries to verify a program. Since function bodies are replaced by their summaries, function summaries greatly simplify the verification task.

However, computing suitable summaries can be difficult particularly at the presence of *recursion*. For non-recursive functions, their summaries can be obtained by inlining function calls. In fact, one can even use summaries of callee functions to compute summaries for non-recursive caller functions. For recursive functions, the depth of recursive calls can be unbounded. Their summaries cannot be obtained by inlining. Moreover, callers and callees can be identical in recursion. Summaries of recursive functions cannot be obtained by summary replacement either.

The traditional approach [1] for finding summaries requires an iterative fixed point computation. Specifically, it iteratively collects the input/output behaviors of a function to construct its function summary until a fixed point is reached. In this paper, we propose an alternative approach that makes use of algorithmic learning. Our basic strategy is *guess-and-check*. We use the CDNF learning algorithm [2], [3] to conjecture candidates of summaries and then verify if the candidate is correct. If a candidate is incorrect, the learning algorithm refines the candidate by posing queries. These queries are answered by a mechanical teacher implemented by a simulator and a SAT solver. Our algorithm is guaranteed to terminate. When the function satisfies the given property, it reports a suitable function summary to justify the correctness. Otherwise, it reports that the function is incorrect.

Compared with traditional approaches, our learning-based technique admits more room for heuristic optimization. Given a program property, there can be several feasible function summaries to establish the property. It suffices to find but one feasible function summary. Our learning-based approach aims to find such a feasible function summary. When the learning algorithm makes a reasonable guess, we check if the guess is a feasible function summary. If it is, we are done. Otherwise, we guide the learning algorithm to find another reasonable guess. Better heuristics can guide the algorithm to better guesses and improve our technique.

We compare our prototype tool with a mature model checker Moped2 [4] for recursive functions on five classical examples with recursion. Our approach can handle three of the five examples with 32-bits variables in seconds, while Moped2 cannot handle any of them.

Related Works: Boolean learning algorithms have been applied to other problems such as loop invariant generation [5], [6], [7], termination analysis [8], and contextual assumption synthesis [9], [10], [11]. In another track, automata-based learning also have applied for similar problems such as contextual assumption synthesis [12], [13], [14] and interface synthesis [15]. Among those, the closest research topic is invariant generation for loops annotated with pre- and post-conditions [5], [6], [7]. Instead of simple loops, we consider a more general model that allows even mutual recursive functions in the current work.

Verification algorithms for recursive programs have been investigated for years. The approach based on collecting summaries was proposed in [1]. Another important technique for verifying recursive programs uses automata to collect reachable system states [16], [17], [18]. Nowadays, several model checkers support recursive Boolean programs. The most well-known and mature tools include Bebop [19] and Moped [4]. More recently, people have tried to use Craig interpolation as the tool for guessing suitable summaries [20], [21]. These works have similar basic strategy to ours, but using completely different techniques.

II. Preliminary

Let $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$ be the *Boolean domain* with *truth values* false and true. We use $\mathbb{B}^n$ to denote the set of all vectors of truth values of size n. A *Boolean variable* is a variable over Boolean domain and a *Boolean formula* is a

formula constructed from Boolean variables, truth values, and Boolean connectives. Let $\bar{x}$ be a finite set of Boolean variables. We use $|\bar{x}|$ to denote the size of $\bar{x}$. An *assignment* of $\bar{x}$ is a mapping from $\bar{x}$ to $\mathbb{B}^{|\bar{x}|}$. We use $x_1, x_2, \ldots x_n \mapsto v_1, v_2, \ldots, v_n$ to denote an assignment that maps x_i to v_i for $0 < i \leq n$. We call the set $\{x_1, x_2, \ldots x_n\}$ the *domain* of the assignment $x_1, x_2, \ldots x_n \mapsto v_1, v_2, \ldots, v_n$.

For an assignment Ψ and a set of variables $\bar{x}$, we write $\Psi \downarrow_{\bar{x}}$ to denote the *projection* of Ψ on $\bar{x}$. We write $\mathsf{true}(\bar{x})$ to denote the assignment that maps all variables in $\bar{x}$ to true. Given a Boolean formula e over Boolean variables $\bar{x}$, we define $[\![e]\!]$ as the set of *satisfying assignments* to e, i.e., $[\![e]\!] = \{\Psi \mid \Psi \models e \text{ and } \Psi \text{ is an assignment to } \bar{x}\}$. A Boolean formula e is *satisfiable* if $[\![e]\!] \neq \emptyset$. Otherwise, we say that e is *unsatisfiable*.

We use $e[x \mapsto x']$ to denote the formula obtained by substituting all free occurrences of x in e with x' and write $e[v, \bar{v} \mapsto v', \bar{v}']$ for $e[v \mapsto v'][\bar{v} \mapsto \bar{v}']$. Given two assignments Ψ and Π, $\Psi[\Pi]$ is the assignment that (1) $\Psi[\Pi](x) = \Pi(x)$ if x in the domain of Π and (2) $\Psi[\Pi](x) = \Psi(x)$ if x is not in the domain of Π. Given a Boolean formula e and an assignment Ψ that gives values to all variables in e, $\{e\}_\Psi$ denotes the *evaluation* of e under Ψ.

A. Control Flow Graphs

A *control flow graph (CFG)* C is a tuple $(V, E, n^{in}, n^{out}, \bar{z}, \bar{y}, \bar{r})$ consists of a set V of *nodes*, a set $E \subseteq V \times \mathsf{op} \times V$ of *edges*, an *entry* node n^{in}, an *exit* node n^{out}, a vector $\bar{z}$ of all variables, a vector $\bar{y} \subset \bar{z}$ of input variables, and a vector $\bar{r} \subset \bar{z}$ of return variables. The set of operations op is generated by the following grammar.

$$\mathsf{op} \equiv x := e \mid e_0 = e_1 \mid \bar{x} := f(\bar{e}) \mid \mathsf{return}\ \bar{r} \mid \mathit{skip}$$

The entry node n^{in} is the only node without incoming edges and the exit node n^{out} is the only node without outgoing edges. The initial values of formal parameters $\bar{y}$ are not determined to represent arbitrary inputs to the function. The initial values of all other variables are true.

We assume the following properties about CFGs:

1) A CFG is a directed acyclic graph (DAG).
2) Each node has at most two outgoing transitions. When it has two outgoing transitions, the two transitions must be labeled by $e_0 = e_1$ and $e_0 = \neg e_1$, respectively, for some e_0, e_1.
3) The exit node n^{out} has only one incoming edge and that edge is the only edge in a CFG labeled with a return operation.
4) If a node has multiple incoming edges, these edges have to be labeled by skip operations.
5) Formal parameters $\bar{y}$ and return variables $\bar{r}$ are disjoint.

B. Semantics of Control Flow Graphs

In the rest of the paper, we fix a CFG $C = (V, E, n^{in}, n^{out}, \bar{z}, \bar{y}, \bar{r})$ obtained from the function f. We use $\emptyset$ to denote the empty stack. If Γ is a stack, then $a \cdot \Gamma$ is a stack with the element a on top of Γ. A *program state* is a triple (Ψ, n, Γ) of an assignment Ψ, a node n, and a call stack Γ. A *call stack* is a stack whose elements are triples of the form $(n, \Psi^c, \bar{x})$, where n is a return node in the calling function, Ψ^c is an assignment to the variables of the calling function, and $\bar{x}$ is a vector of variables to receive return values.

The semantic of C is defined by its induced transition system over program states. The *initial program states* are $\{(\Psi^i, n^{in}, \emptyset) \mid \Psi^i = \mathsf{true}(\bar{z})[\bar{y} \mapsto \bar{v}] \wedge \bar{v} \in \mathbb{B}^{|\bar{y}|}\}$. In words, Ψ^i is an assignment that maps formal parameters $\bar{y}$ to arbitrary Boolean values and the rest to true. Define the Post (partial) function mapping a program state and an edge to the next program state as follows.

$$\begin{aligned}
\mathit{Post}((\Psi, n, \Gamma), (n, x := e, n')) &\triangleq \\
&(\Psi[x \mapsto \{e\}_\Psi], n', \Gamma) \\
\mathit{Post}((\Psi, n, \Gamma), (n, e_0 = e_1, n')) &\triangleq \\
&(\Psi, n', \Gamma) \ \ \textbf{if} \ \ \{e_0\}_\Psi = \{e_1\}_\Psi \\
\mathit{Post}((\Psi, n, \Gamma), (n, \bar{x} := f(\bar{e}), n')) &\triangleq \\
&(\mathsf{true}(\bar{z})[\bar{y} \mapsto \{\bar{e}\}_\Psi], n^{in}, (n', \Psi, \bar{x}) \cdot \Gamma) \\
\mathit{Post}((\Psi, n, (n^r, \Psi^r, \bar{x}) \cdot \Gamma), (n, \mathsf{return}\ \bar{r}, n')) &\triangleq \\
&(\Psi^r[\bar{x} \mapsto \{\bar{r}\}_\Psi], n^r, \Gamma) \\
\mathit{Post}((\Psi, n, \Gamma), (n, \mathit{skip}, n')) &\triangleq (\Psi, n', \Gamma)
\end{aligned}$$

For other cases, $\mathit{Post}((\Psi, n, \Gamma), (n_1, op, n_2)) = \bot$. We say C returns $\bar{v}' \in \mathbb{B}^{|\bar{r}|}$ on the input $\bar{v} \in \mathbb{B}^{|\bar{y}|}$ if there are $(\Psi_1, n_1, \Gamma_1), op_1, (\Psi_2, n_2, \Gamma_2), \ldots, (\Psi_k, n_k, \Gamma_k)$ with $n_1 = n^{in}$, $n_k = n^{out}$, $\bar{v} = \{\bar{y}\}_{\Psi_1}$, $\bar{v}' = \{\bar{r}\}_{\Psi_k}$, and $\mathit{Post}((\Psi_i, n_i, \Gamma_i), op_i) = (\Psi_{i+1}, n_{i+1}, \Gamma_{i+1})$ for $0 < i < k$.

C. Boolean Programs

We consider *Boolean programs* with syntax as follows*.

$$\begin{aligned}
\mathit{prog} &\equiv \mathit{func}^+ \\
\mathit{func} &\equiv \mathit{name}(\bar{y})\{s^*\} \\
s &\equiv x := e \mid \bar{x} := f(\bar{e}) \mid \mathsf{return}\ \bar{r} \mid \\
&\quad\ \mathsf{if}\ e_0 = e_1\ \mathsf{then}\ s^*\ \mathsf{else}\ s^* \mid \\
&\quad\ \mathsf{while}\ e_0 = e_1\ \mathsf{do}\ s^*\ \mathsf{od}
\end{aligned} \tag{1}$$

where x is a Boolean variable, e, e_0, e_1 are Boolean formulae, $\bar{e}$ is a vector of Boolean formulae, f is a function, $\bar{x}$ is a vector of variables to receive the return values, and $\bar{r}$ a vector of return variables for functions.

A Boolean program can be translated to a CFG. The translation involves the following steps (1) transforms all loops to recursive functions, (2) inlines all non-recursive functions, (3) merges all recursive functions into one, and

*The first function is the starting point of a Boolean program.

(4) makes the return $\bar{r}$ statement the last statement of the program. For example, consider a simple function with a loop in Figure 1 (a). We introduce a new function for the loop body. A recursive call to the introduced function thus performs an iteration of the loop (Figure 1 (b)).

```
A(x, y){
  while x do
    x:=~y;
    y:=~x;
  od
  return x;
}
```

(a) Original

```
A(x, y){
  x,y:=doLoop(x,y);
  return x;
}
doLoop(x, y){
  if x then
    x:=~y;
    y:=~x;
    x,y:=doLoop(x,y);
    return x,y;
  else
    return x,y;
}
```

(b) After transformation

Figure 1: Encode loops as recursive functions.

After the transformation, Boolean programs are converted into control flow graphs easily. For the if ... then ... else statements, we convert them to nodes with two outgoing transitions. This is the only case that a node can have more than one outgoing transition. For other statements, each statement is converted into one edge of the CFG connected in the same order as in the Boolean program. If a node n has multiple incoming edges, e.g., (n_1, op_1, n) and (n_2, op_2, n), we add edges labeled $skip$ in the middle, e.g., replace the edges with the (n_1, op_1, n_1') and $(n_1', skip, n)$, (n_2, op_2, n_2'), and $(n_2', skip, n)$. Notice that after the transformation, the Boolean program contains only one function. Moreover its corresponding CFG satisfies the assumptions.

D. Function Summary

A *function summary* is an over-approximation of a function's input/output behaviors. Formally, a function summary s is a formula over $\bar{y} \cup \bar{r}$ (formal parameters and return variables) such that if C returns $\bar{v}' \in \mathbb{B}^{|\bar{r}|}$ on the input $\bar{v} \in \mathbb{B}^{|\bar{y}|}$, then $\{s\}_{\bar{y},\bar{r} \mapsto \bar{v},\bar{v}'} = \mathsf{true}$.

E. Problem Statement

Given a Boolean formula p over $\bar{y} \cup \bar{r}$ that specifies the desired behaviors, the function f is *correct* with respect to p if f always produces input/output behaviors specified in p. This is done by finding a summary s of the function f such that $s \wedge \neg p$ is unsatisfiable. Indeed, suppose there is a summary s with $s \wedge \neg p$ unsatisfiable. We have $s \implies p$. Since s is a summary of f, every behaviors of f are specified in s. f is correct with respect to p.

F. The CDNF Learning Algorithm

We use the CDNF algorithm to find a proper function summary. The learning algorithm was proposed in [2]. It assumes a *teacher* that knows the target Boolean formula t and answers the following two types of questions about t:

- A *membership query* on an assignment Ψ asks if the target Boolean formula t evaluates to true under Ψ.
- An *equivalence query* on a candidate Boolean formula s asks if s is equivalent to t. If they are not, a differentiating assignment is returned as a counterexample.

Let t be the target formula. Let $DNF(t)$ and $CNF(t)$ be the sizes of minimal DNF and CNF formulae equal to t, respectively. If a teacher of t is provided, the CDNF algorithm is guaranteed to produce a formula that equals to t using a polynomial number of both type of queries (in $DNF(t)$ and $CNF(t)$). We treat the CDNF algorithm as a black box here and refer interested readers to [2], [3].

III. Overview

We apply the CDNF machine learning algorithm to find a proper function summary [2], [3]. The CDNF learning algorithm acquires information about a function summary by posing *membership queries* to the teacher. A *membership query* checks if an assignment Ψ corresponds to a real input/output behavior of f. After collecting a number of membership query results, the CDNF algorithm conjectures a candidate of a function summary s. It then poses an *equivalence query* to check if s is indeed a function summary that is sufficient to establish correctness. If it is not, the teacher either reports that the program is unsafe or returns an assignment Φ that can be used by the CDNF algorithm to refine the next candidate. The above procedure is repeated until a function summary is found or the algorithm detects that f is incorrect with respect to the desired property.

Notice that when the function f is correct with respect to a property p, it is usually the case that there are many function summaries that can be used to establish the correctness proof. We would like to apply the CDNF learning algorithm to find *one* of such summaries to prove program correctness.

Let s_f be the *minimal function summary* for f[†], that is, the function summary of f with the smallest amount of satisfying assignments. Our design of the teacher guides the CDNF algorithm to the Boolean formula $s_f \wedge p$. If f satisfies p, that is, $[\![s_f]\!] \subseteq [\![p]\!]$, we have $[\![s_f \wedge p]\!] = [\![s_f]\!] \cap [\![p]\!] = [\![s_f]\!]$. Hence the CDNF algorithm eventually converges to $[\![s_f]\!]$ and finds a function summary. Nevertheless, it is very often that our algorithm finds a feasible summary before it converges to $[\![s_f]\!]$. If f does not satisfy p, that is, $s_f \not\implies p$, our teacher eventually reports an assignment Φ in $[\![s_f \wedge \neg p]\!]$ as an evidence for f violating the property p (Figure 2).

[†]In fact, s_f equals to the exact semantics of the function f [22].

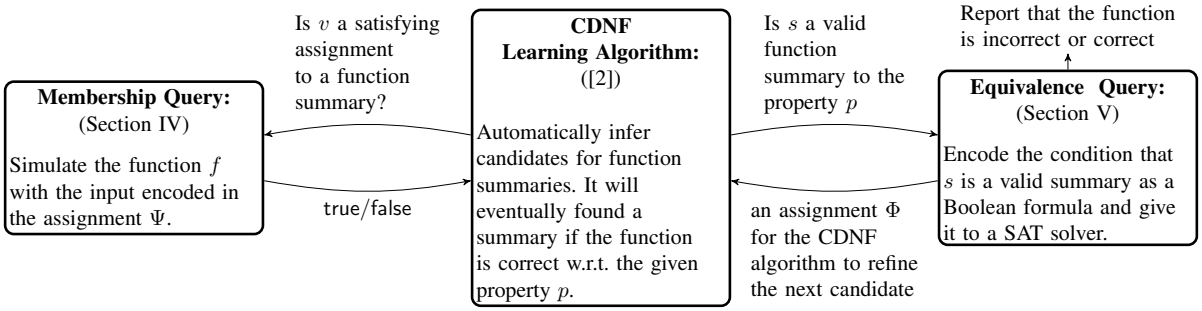

Figure 2: Components of our framework

The design of the teacher for membership and equivalence queries will be described in Section IV and V. With the teacher provided, our procedure is guaranteed to terminate.

IV. Answering Membership Queries

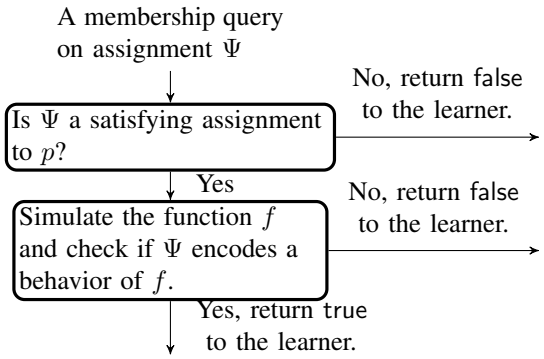

Figure 3: The flow to answer membership queries

On a membership query, the teacher receives an assignment Ψ over $\bar{y} \cup \bar{r}$. It first checks if Ψ is a satisfying assignment to p. If it is not, Ψ cannot be a satisfying assignment in $[\![s_f]\!] \cap [\![p]\!]$. The teacher thus returns false to the learning algorithm (Figure 3).

Otherwise, the teacher simulates the function f on the assignment Ψ. The assignment Ψ is divided into an assignment to formal parameters $\Psi \downarrow_{\bar{y}}$, and an assignment to return variables $\Psi \downarrow_{\bar{r}}$. The simulation algorithm is almost equivalent to executing the program on the input $\Psi \downarrow_{\bar{y}}$ and checking if the output matches $\Psi \downarrow_{\bar{r}}$. The only difference is that we reject the non-terminating inputs in simulation. More specifically, once the function f calls itself, we record the assignment Π to its formal parameters in a stack. The simulator stops immediately and returns false if this assignment Π appeared in the stack already. In this case, inputting Π to f will eventually lead to another call to f with the same input. The computation will be repeated forever. The function f hence has no return values. This is the only possible situation that the execution of f does not terminate.

Input: An assignment Ψ to formal parameters and return variables

1 $(\Pi, n, \Gamma) := (\mathsf{true}(\bar{z})[\Psi \downarrow_{\bar{y}}], n^{in}, \emptyset)$; $VStack := \emptyset$;
2 **while** true **do**
3 **if** $\exists (n, \mathsf{op}, n') \in E.Post((\Pi, n, \Gamma), (n, \mathsf{op}, n')) \neq \bot$ **then**
4 **if** op *is* $\bar{x} := f(\bar{e})$ **then**
5 **if** $\{\bar{e}\}_\Pi \in VStack$ **then return** false ;
6 **else** Push $\{\bar{e}\}_\Pi$ to $VStack$;
7 **if** op *is* return $\bar{r}$ **then**
8 **if** $VStack = \emptyset$ **then**
9 **if** $\Pi \downarrow_{\bar{r}} = \Psi \downarrow_{\bar{r}}$ **then return** true;
10 **else return** false;
11 **else** Pop the top of $VStack$;
12 $(\Pi, n, \Gamma) := Post((\Pi, n, \Gamma), (n, \mathsf{op}, n'))$;
13 **else return** false;
14 **end**

Algorithm 1: Simulate the function f on Ψ.

Algorithm 1 gives details of our simulator. In Line 1, variables are initialized. We use Π, n, and Γ to denote the assignment, the node, and the call stack of the current program state, respectively. A stack $VStack$ is used to store the values that have been assigned to the formal parameters. In Lines 3-12, we compute the next program state. If such program state does not exist (Line 13), the simulation stops and returns false. Lines 4-6 handle function calls. If the values $\{\bar{e}\}_\Pi$ appeared in $VStack$ (Line 5), the function does not terminate on this input and hence the simulator returns false. Lines 7-11 handle return operations. If $VStack$ is not empty (Line 11), an element is popped from $VStack$. Otherwise (Line 8), the simulation stops and $\Pi \downarrow_{\bar{r}}$ is the return value of f on the input $\Psi \downarrow_{\bar{y}}$. When $\Pi \downarrow_{\bar{r}}$ are equal to the return values $\Psi \downarrow_{\bar{r}}$, Ψ encodes a behavior of f and hence the teacher returns true. Otherwise, false is returned.

V. Answering Equivalence Queries

On an equivalence query, the teacher is given a Boolean formula s over $\bar{y} \cup \bar{r}$. We check if $[\![s]\!] \subseteq [\![p]\!]$ by checking

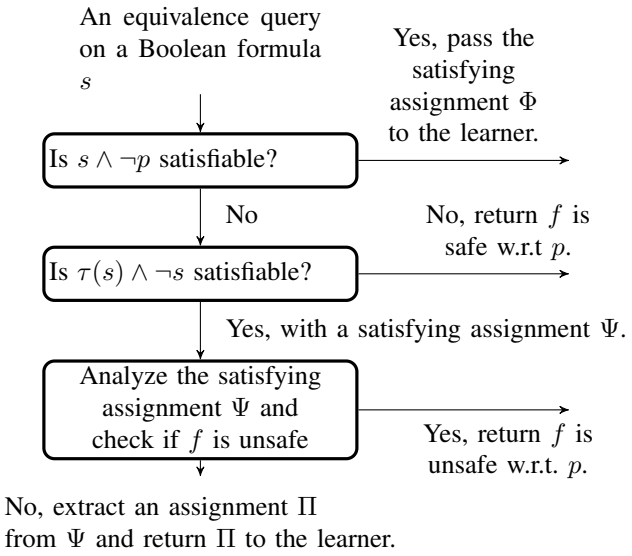

Figure 4: The flow to answer equivalence queries

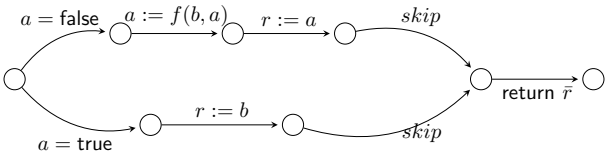

Figure 5: A control flow graph

the satisfiability of the Boolean formula $s \wedge \neg p$. A satisfying assignment Φ of $s \wedge \neg p$ denotes an input/output behavior allowed by the current candidate s (i.e., $\Phi \in [\![s]\!]$), but not allowed in p (i.e., $\Phi \notin [\![p]\!]$) (Figure 4). Therefore, we pass Φ to the CDNF algorithm to refine the next candidate.

Otherwise, we proceed to check if s is a summary of f. Intuitively, we have to generate the formula s_f that describes exact input/output behaviors of f and check if $s_f \implies s$, i.e, s encodes all possible behaviors of the function f. However, it is hard to obtain s_f in particular when f is a recursive function. Hence we adopt a slightly different approach. The theory behind the approach is Tarski's fixed point theorem.

Let τ be a functional that takes a Boolean formula s over $\bar{y} \cup \bar{r}$ as the input, and returns a Boolean formula encoding the behaviors of f assuming s is a function summary of f. It can be shown that the smallest function summaries of f is the least fixed point for τ. By Tarski's Fixed Point Theorem,

$$\mu z.\tau(z) = \wedge\{z : \tau(z) \implies z\}.$$

It follows that any formula z satisfies $\tau(z) \implies z$ encodes a function summary of f.

After the teacher checks $s \implies p$ in Figure 4, it continues to check if $\tau(s) \implies s$. If $\tau(s) \wedge \neg s$ is unsatisfiable, s is a function summary that can be used to establish correctness with respect to p. Hence our algorithm terminates and reports that f is safe with respect to p. If $\tau(s) \wedge \neg s$ is satisfied, we analyze the satisfying assignment as follows.

Let Ψ be a satisfying assignment to $\tau(s) \wedge \neg s$. There are only two possible reasons that the $\tau(s) \wedge \neg s$ is satisfiable. Either s encodes too few behaviors, or $\tau(s)$ encodes too many behaviors. We use simulation (Algorithm 1) to check if $\Psi \downarrow_{\bar{y},\bar{r}}$ is indeed a behavior of f. If so, define Π to be $\Psi \downarrow_{\bar{y},\bar{r}}$. Thus $\Pi \in [\![s_f]\!]$ but $\Pi \notin [\![s]\!]$. If moreover $\Pi \in [\![\neg p]\!]$, Π is a behavior of f not satisfying p. The teacher reports f is unsafe with respect to p. If $\Pi \in [\![p]\!]$, s does not encode the behavior $\Pi \in [\![s_f]\!] \cap [\![p]\!]$. The teacher returns Π as a counterexample to refine the next candidate.

Otherwise, Π does not encode a behavior of f and hence the function summary s is too coarse. Some assignments thus need be removed from $[\![s]\!]$. We again use Algorithm 1 to check if a satisfying assignment $\hat{\Psi}$ in $[\![s]\!]$ is a behavior of f. If not, we have $\hat{\Psi} \in [\![s]\!]$ but $\hat{\Psi} \notin [\![s_f]\!] \cap [\![p]\!]$. $\hat{\Psi}$ is returned as a counterexample to the learning algorithm.

VI. Correctness

Termination: We first assume that our algorithm does not terminate. By the property of CDNF, the teacher eventually conjectures a candidate formula s that equals to $s_f \wedge p$. After this candidate, we have two cases: (1)$[\![s]\!] = [\![s \wedge p]\!] = [\![s_f]\!]$ and (2) $[\![s]\!] = [\![s_f \wedge p]\!] \subset [\![s_f]\!]$. For case (1), because $[\![s]\!] = [\![s_f]\!] = [\![s_f \wedge p]\!] = [\![s \wedge p]\!]$, we know that $s \implies p$ is valid. Because $[\![s]\!] = [\![s_f]\!]$, the smallest function summary, we have $\tau(s) \implies s$ is valid. Since the two conditions $s \implies p$ and $\tau(s) \implies s$ are both valid, the algorithm will terminate and answer that the function f is safe with respect to p. For case (2), $[\![s]\!] \subset [\![s_f]\!]$, there must be some assignment Ψ such that $\Psi \in [\![s_f]\!]$ and $\Psi \notin [\![s]\!] = [\![s_f \wedge p]\!]$. It follows that $\Psi \in [\![s_f \wedge \neg p]\!]$. In this case, our algorithm will stop and then report that f is unsafe with respect to p. From the analysis of case (1) and (2), we proved that our algorithm always terminates.

Soundness: We next show that when our algorithm terminates, it always gives a correct answer. For the case that it found a summary s, the two conditions $s \implies p$ and $\tau(s) \implies p$ are both valid and hence we proved that the function is safe with respect to p. For the case that the teacher found an assignment Φ in $[\![s_f]\!]$ but not in $[\![p]\!]$, Φ is an evidence to show that f is unsafe with respect to p and thus we proved that the function is unsafe.

VII. Optimization

In this section we explain how to translate a control flow graph to a path formula efficiently.

Given a CFG C, we first translate the statement on each edge to the static single assignment form (Figure 6). In the figure, we add indices to variables to make data flow explicit. For instance, the return variable r have different indices on different edges. This allows us to form a formula for each

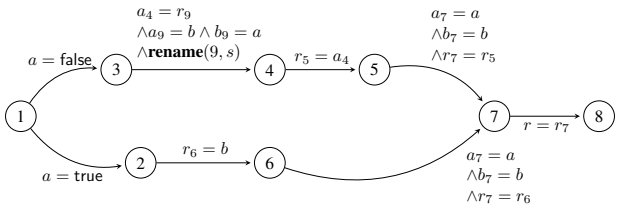

Figure 6: Static Single Assignments of Figure 5.

path. The path $1-2-6-7-8$ is semantically equivalent to the formula $a = \mathsf{true} \wedge r_6 = b \wedge (a_7 = a \wedge b_7 = b \wedge r_7 = r_6) \wedge r = r_7$. The function **rename**(i, s) in the figure takes a formula s and renames every variables in s by adding the index i. The formula $a_4 = r_9 \wedge a_9 = b \wedge b_9 = a \wedge$ **rename**$(9, s)$ on the edge $3-4$ hence models the function call $a := f(b, a)$ when s is a function summary of f.

Assume s is a function summary of f. The path formula $\tau(s)$ of the CFG C can be obtained by the disjunction of path formulae over all paths in C. Although it is logically sound, such path formulae may contain redundant subformulae. If one computes such a path formula from Figure 6, the subformula $r = r_7$ on the edge $7-8$ will appear twice. Complicated control flow graphs will have lots of repeated subformulae. Simplifying $\tau(s)$ by sharing is hence needed.

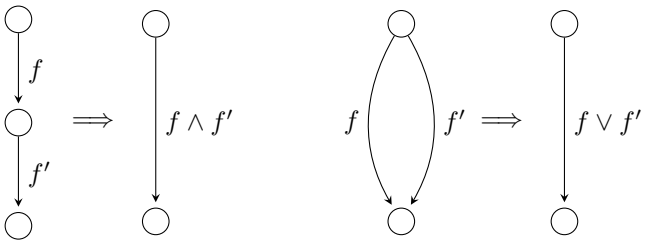

Figure 7: Merge edges in C_{SSA}.

Figure 7 enables sharing in path formulae by graph transformation. Our algorithm merges two edges and their formulae in a control flow graph by the figure. Using graph transformation, the path formula $\tau(s)$ for the control flow graph in Figure 6 contains only one occurrence of the subformula $r = r_7$ as desired.

VIII. Preliminary Experimental Results

We implemented our algorithm in C++ and compared it with the Moped2 model checker [4]. We did not enable predicate abstraction. We aim for testing the core for state space exploration algorithm in the experiments.

We encode five classical examples of recursive functions of n-bits variables (n varies from 3 to 32): $gcd(\bar{x}, \bar{y})$, $hanoi(\bar{n})$[‡], $fibonacci(\bar{n})$, $combination(\bar{n}, \bar{k})$, and $sum(\bar{n})$ against different properties. These examples contain arithmetic operations, which often require complicated function summaries for proving their properties. Therefore, we think our examples are more difficult for summary generation than Boolean programs generated from predicate abstraction. Moreover, these examples are more suitable to test scalability. We can easily obtain bigger and bigger CFG by increasing the bit-width. In the current experiment, we have examples from 4bits to 32bits. The latter normally contain a few hundreds of Boolean variables.

We use the CDNF learning algorithm implemented in the BULL library [23]. We use minisat2 for SAT solving. All experiments are done on AMD Opteron(TM) 6276 CPUs with 256GB memory running Linux 3.4.7. The timeout period and memory limit for each process are set to 300 seconds and 4 GB, respectively. Observe that our approach relies heavily on a simulator. Its implementation is crucial to the performance. We add a cache to the simulator. The cache hits rate is normally ranging from 60% to 90%.

We use $\bar{r}$ to denote the result returned from a function. For each case, we test three positive properties: (1) zero test ($\bar{r} > 0$), (2) a given input produces correct output (e.g., for $gcd(\bar{x}, \bar{y})$, we check $\bar{x} \neq 3 \vee \bar{y} \neq 2 \vee \bar{r} = 1$), and (3) a problem specific difficult property (e.g., for $gcd(\bar{x}, \bar{y})$, we check the property that two even input values will not result in an odd output value). In the CFGs of all the examples, we put a filter to exclude the input value 0. Therefore, in all of the examples, the verifier should report "f is safe with respect to the zero test".

We present the experimental result in Table I. Our algorithm performs well on examples $hanoi(\bar{n})$, $fibonacci(\bar{n})$, and $sum(\bar{n})$. It verifies all properties with almost no time even with 32 bits variables. For the 32 bits examples and property (3), the CDNF algorithms poses 2 membership queries and 5 equivalence queries to find a summary for $hanoi(\bar{n})$, 63 membership queries and 63 equivalence queries to find a summary for $fibonacci(\bar{n})$, and 64 membership queries and 63 equivalence queries to find a summary for $sum(\bar{n})$. We attribute this to the fact that usually there exist many function summaries for f to satisfy the property p and we only need to find *one* of them. However, the learned summaries need not be trivial. Many learned function summaries are Boolean formulae with hundreds of literals. In contrast, the Moped2 tool cannot handle any of these examples with variable size larger than 16 bits.

However, our tool did not get good results on $gcd(\bar{x}, \bar{y})$ and $combination(\bar{n}, \bar{k})$. After a closer scrutiny, we found that in both examples, when we add an unnecessary assignment to the summary, it will take a huge amount of time to propagate the information and reach a new fixed point. For example, if we add $gcd(1, 100) = 10$ to the summary, even if this assignment is not essential with respect to the properties, the algorithm has to also add $gcd(1, 101) = 10$, $gcd(101, 100) = 10$, $gcd(1, 102) = 10, \ldots$ to the summary

[‡]It counts the number of moves. That is, $hanoi(\bar{n}) = 2 \times hanoi(\bar{n} - 1) + 1$.

Table I: The experimental result on the classical examples with different variable sizes ('-' denotes timeout).

$fibonacci(\bar{n})$

	Property	4bits	8bits	12bits	16bits	20bits	24bits	28bits	32bits
Ours	$\bar{r} > 0$	0.01	0.04	0.07	0.11	0.16	0.22	0.29	0.37
Moped2	$\bar{r} > 0$	0	1.26	-	-	-	-	-	-
Ours	$\bar{n} \neq 4 \vee \bar{r} = 5$	0.01	0.04	0.07	0.11	0.16	0.26	0.29	0.37
Moped2	$\bar{n} \neq 4 \vee \bar{r} = 5$	0	1.26	-	-	-	-	-	-
Ours	$\bar{n} \leq 3 \vee \bar{r} \neq \bar{n}$	0.03	0.11	0.28	0.55	0.97	1.5	2.19	3.07
Moped2	$\bar{n} \leq 3 \vee \bar{r} \neq \bar{n}$	0	2.1	-	-	-	-	-	-

$hanoi(\bar{n})$

	Property	4bits	8bits	12bits	16bits	20bits	24bits	28bits	32bits
Ours	$\bar{r} > 0$	0.01	0.01	0.02	0.02	0.03	0.04	0.05	0.06
Moped2	$\bar{r} > 0$	0	0.02	2.37	-	-	-	-	-
Ours	$\bar{n} \neq 3 \vee \bar{r} = 7$	0.01	0.01	0.02	0.02	0.03	0.04	0.05	0.07
Moped2	$\bar{n} \neq 3 \vee \bar{r} = 7$	0	0.02	2.36	-	-	-	-	-
Ours	$\bar{n} = 1 \vee \bar{r} \neq \bar{n}$	0.01	0.02	0.02	0.03	0.04	0.06	0.07	0.08
Moped2	$\bar{n} = 1 \vee \bar{r} \neq \bar{n}$	0	0.78	-	-	-	-	-	-

$gcd(\bar{x}, \bar{y})$

	Property	4bits	8bits	12bits	16bits	20bits	24bits	28bits	32bits
Ours	$\bar{r} > 0$	2.78	-	-	-	-	-	-	-
Moped2	$\bar{r} > 0$	0	3.38	-	-	-	-	-	-
Ours	$\bar{x} \neq 3 \vee \bar{y} \neq 2 \vee \bar{r} = 1$	2.32	-	-	-	-	-	-	-
Moped2	$\bar{x} \neq 3 \vee \bar{y} \neq 2 \vee \bar{r} = 1$	0	3.37	-	-	-	-	-	-
Ours	$\bar{x}\%2{=}1 \vee \bar{y}\%2{=}1 \vee \bar{r}\%2{=}0$	3.47	-	-	-	-	-	-	-
Moped2	$\bar{x}\%2{=}1 \vee \bar{y}\%2{=}1 \vee \bar{r}\%2{=}0$	0	5.17	-	-	-	-	-	-

$sum(\bar{n})$

	Property	4bits	8bits	12bits	16bits	20bits	24bits	28bits	32bits
Ours	$\bar{r} > 0$	0.01	0.01	0.02	0.03	0.05	0.06	0.08	0.1
Moped2	$\bar{r} > 0$	0	1.23	-	-	-	-	-	-
Ours	$\bar{n} \neq 3 \vee \bar{r} = 6$	0.01	0.01	0.02	0.03	0.05	0.06	0.08	0.1
Moped2	$\bar{n} \neq 3 \vee \bar{r} = 6$	0	1.24	-	-	-	-	-	-
Ours	$\bar{n} = 1 \vee \bar{r} \neq \bar{n}$	0.01	0.05	0.16	0.22	0.35	0.54	0.77	1.04
Moped2	$\bar{n} = 1 \vee \bar{r} \neq \bar{n}$	0	2.09	-	-	-	-	-	-

$combination(\bar{n}, k)$

	Property	4bits	8bits	12bits	16bits	20bits	24bits	28bits	32bits
Ours	$\bar{r} > 0$	26.16	-	-	-	-	-	-	-
Moped2	$\bar{r} > 0$	0	130.1	-	-	-	-	-	-
Ours	$\bar{n} \neq 4 \vee \bar{k} \neq 2 \vee \bar{r} = 6$	27.28	-	-	-	-	-	-	-
Moped2	$\bar{n} \neq 4 \vee \bar{k} \neq 2 \vee \bar{r} = 6$	0.01	131.04	-	-	-	-	-	-
Ours	$\bar{n} \neq \bar{k} \vee \bar{r} = 1$	27.75	-	-	-	-	-	-	-
Moped2	$\bar{n} \neq k \vee \bar{r} = 1$	0.01	132.42	-	-	-	-	-	-

until reached the maximal values of $\bar{x}$ and $\bar{y}$. A similar phenomena can be observed in the case of $combination(\bar{n}, \bar{k})$. We believe that techniques such as widening could be useful here to speed up the convergence for these cases.

IX. Concluding Remarks

We proposed a sound and complete approach for finding proper function summary to prove a given property p based on learning. The experimental results on classical examples are encouraging. We believe that the approach can be made practical with further fine tune. As an initial attempt of applying learning for finding function summaries, our implementation is still primitive, e.g., we support only Boolean type and hence it is very tedious to encode examples. As the next step, we plan to integrate it with the front-end of established tools such as Bebop [19] and Moped [4].

One of the benefit of our approach is that it admits *incremental learning*. Based on a summary s that can be used to prove p, one can continue the execution of our approach to find a summary for a tighter property $p \wedge p'$. Therefore it allows the scenario that when combining it

with program verification use proof assistants, one may first replace all functions with the learned summaries. When the summaries are too weak to prove the final property, one can try to derive more proof obligations and continue using our approach to refine the summaries.

In order to achieve better scalability, we believe that the use of abstraction is necessary. We have thought about combining our technique with predicate abstraction [24]. To be more specific, we can use predication abstraction to obtain a Boolean program [25] that describes an over-approximation of the original program's behaviors. However, our technique uses a simulator to answer queries. Boolean programs obtained from predicate abstraction requires Boolean variables with don't care values. These don't care values can induce *non-determinism* and hence cannot be handle by simulation. We have to modify our simulation algorithm, perhaps to use a symbolic one, in order to handle don't care values.

ACKNOWLEDGMENT

This work is partially supported by the Ministry of Science and Technology of Taiwan under grant numbers 103-2221-E-001 -020 -MY3 and 103-2221-E-001 -019 -MY3.

REFERENCES

[1] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995, pp. 49–61.

[2] N. H. Bshouty, "Exact learning Boolean function via the monotone theory," *Information and Computation*, vol. 123, no. 1, pp. 146–153, 1995.

[3] Y.-F. Chen and B.-Y. Wang, "Learning boolean functions increamentally," in *CAV*, ser. LNCS. Springer, 2012, pp. 55–70.

[4] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," in *CAV*, 2001, pp. 324–336.

[5] Y. Jung, S. Kong, B.-Y. Wang, and K. Yi, "Deriving invariants in propositional logic by algorithmic learning, decision procedure, and predicate abstraction," in *VMCAI*, ser. LNCS. Springer, 2010, pp. 180–196.

[6] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi, "Automatically inferring quantified loop invariants by algorithmic learning from simple templates," in *APLAS*, ser. LNCS, vol. 6461. Springer, 2010, pp. 328–343.

[7] Y. Jung, W. Lee, B.-Y. Wang, and K. Yi, "Predicate generation for learning-based quantifier-free loop invariant inference," in *TACAS*, ser. LNCS, vol. 6605. Springer, 2011, pp. 205–219.

[8] W. Lee, B.-Y. Wang, and K. Yi, "Termination analysis with algorithmic learning," in *CAV*, 2012, pp. 88–104.

[9] Y.-F. Chen, E. M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang, "Automated assume-guarantee reasoning through implicit learning," in *CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 511–526.

[10] Y.-F. Chen, E. M. Clarke, A. Farzan, F. He, M.-H. Tsai, Y.-K. Tsay, B.-Y. Wang, and L. Zhu, "Comparing learning algorithms in automated assume-guarantee reasoning," in *ISoLA (1)*, ser. LNCS, vol. 6415. Springer, 2010, pp. 643–657.

[11] F. He, B.-Y. Wang, L. Yin, and L. Zhu, "Symbolic assume-guarantee reasoning through bdd learning," in *ICSE*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 1071–1082.

[12] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *TACAS*, ser. LNCS, vol. 2619. Springer, 2003, pp. 331–346.

[13] M. G. Bobaru, C. S. Păsăreanu, and D. Giannakopoulou, "Automated assume-guarantee reasoning by abstraction refinement," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 135–148.

[14] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu, "Proof rules for automated compositional verification through learning," in *Workshop on Specification and Verification of Component-Based Systems*, 2003, pp. 14–21.

[15] R. Singh, D. Giannakopoulou, and C. S. Pasareanu, "Learning component interfaces with may and must abstractions," in *CAV*, ser. Lecture Notes in Computer Science, vol. 6174. Springer, 2010, pp. 527–542.

[16] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," *Electr. Notes Theor. Comput. Sci.*, vol. 9, pp. 27–37, 1997.

[17] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model checking pushdown systems," in *CAV*, 2000, pp. 232–247.

[18] R. Alur, K. Etessami, and M. Yannakakis, "Analysis of recursive state machines," in *CAV*, 2001, pp. 207–220.

[19] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for Boolean programs," in *SPIN*, 2000, pp. 113–130.

[20] O. Sery, G. Fedyukovich, and N. Sharygina, "Interpolation-based function summaries in bounded model checking." in *Haifa Verification Conference*, 2011, pp. 160–175.

[21] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "Whale: An interpolation-based algorithm for inter-procedural verification," in *VMCAI*, 2012, pp. 39–55.

[22] D. Scott and C. Strachey, *Toward a mathematical semantics for computer languages*. Oxford Programming Research Group Technical Monograph, 1971.

[23] Y.-F. Chen and B.-Y. Wang, "BULL: a library for learning algorithms of boolean functions," in *TACAS*, 2013.

[24] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *CAV*, 1997, pp. 72–83.

[25] T. Ball and S. Rajamani, "Boolean programs: A model and process for software analysis," Microsoft Research, Tech. Rep. MSR-TR-2000-14, 2000.