

# Verifying Recursive Programs using Intraprocedural Analyzers

Yu-Fang Chen<sup>1</sup>, Chiao Hsieh<sup>1,2</sup>, Ming-Hsien Tsai<sup>1</sup>, Bow-Yaw Wang<sup>1</sup>, and  
Farn Wang<sup>2</sup>

<sup>1</sup> Institute of Information Science, Academia Sinica, Taiwan

<sup>2</sup> Graduate Institute of Electrical Engineering, National Taiwan University, Taiwan

**Abstract.** Recursion can complicate program analysis significantly. Some program analyzers simply ignore recursion or even refuse to check recursive programs. In this paper, we propose an algorithm that uses a recursion-free program analyzer as a black box to check recursive programs. With extended program constructs for assumptions, assertions, and nondeterministic values, our algorithm computes function summaries from inductive invariants computed by the underlying program analyzer. Such function summaries enable our algorithm to check recursive programs. We implement a prototype using the recursion-free program analyzer CPACHECKER and compare it with other program analyzers on the benchmarks in the 2014 Competition on Software Verification.

## 1 Introduction

Program verification is a grand challenge with significant impact in computer science. Its main difficulty is in great part due to common program features such as concurrent execution, pointers, recursive function calls, and unbounded basic data types [11]. Subsequently, it is extremely complicated to develop a verification algorithm that handles all features. Researches on program verification typically address some of these features and simplify others. Verification tools however are required to support as many features as possible. Since implementation becomes increasingly unmanageable with additional features, incorporating algorithms for all features in verification tools can be a nightmare for developers.

One way to address the implementation problem is by reduction. If verifying a new feature can be transformed to existing features, development efforts can be significantly reduced. In this paper, we propose an algorithm to extend intraprocedure (recursive-free) program analyzers to verify recursive programs. Such analyzers supply an *inductive invariant* when a program is verified to be correct and support program constructs such as assumptions, assertions, and nondeterministic values. Our algorithm transforms any recursive program into non-recursive ones and invokes an intraprocedure program analyzer to verify properties about the generated non-recursive programs. The verification results allow us to infer properties on the given recursive program.

Our algorithm proceeds by iterations. In each iteration, it transforms the recursive program into a non-recursive program that *under-approximates* the

behaviors of the original and sends the under-approximation to an intraprocedure program analyzer. If the analyzer verifies the under-approximation, purported *function summaries* for recursive functions are computed. Our algorithm transforms the original recursive program into more non-recursive programs with purported function summaries. It then checks if purported function summaries are correct by sending these non-recursive programs to the analyzer.

Compared with other analysis algorithms for recursive programs, ours is very lightweight. It only performs syntactic transformation and requires standard functionalities from underlying intraprocedure program analyzers. Moreover, our technique is very modular. Any intraprocedural analyzer providing proofs of inductive invariants can be employed in our algorithm. With the interface between our algorithm and program analyzers described in this work, incorporating recursive analysis with existing program analyzers thus requires minimal implementation efforts. Recursive analysis benefits from future advanced intraprocedural analysis with little cost through our lightweight and modular technique.

We implement a prototype using CPACHECKER (over 200 thousands lines of JAVA codes) as the underlying program analyzer [10]. In our prototype, 1256 lines of OCAML codes are for syntactic transformation and 880 lines of PYTHON codes for the rest of the algorithm. 324 lines among them are for extracting function summaries. Since syntactic transformation is independent of underlying program analyzers, only about 15% of codes need to be rewritten should another analyzer be employed. We compare it with program analyzers specialized for recursion in experiments. Although CPACHECKER does not support recursion, our prototype scores slightly better than the second-place tool ULTIMATE AUTOMIZER on the benchmarks in the 2014 Competition on Software Verification [1].

**Organization:** Preliminaries are given in Section 2. We give an overview of our technique in Section 3. Technical contributions are presented in Section 4. Section 5 reports experimental results. Section 6 describes related works. Finally, some insights and improvements are discussed in Section 7.

## 2 Preliminaries

Let  $\mathbf{Vars}$  denote the set of *program variables* and  $\mathbf{Vars}' = \{x' : x \in \mathbf{Vars}\}$ . We consider a variant of the WHILE language [18]:

Expression $\ni p, q ::= x$	$x \in \mathbf{Vars}$
<b>false</b>   <b>true</b>   0   1   ...	constant
<b>nondet</b>	non-deterministic value
$f(\bar{p})$	function invocation
$p \odot p$	$\odot \in \{+, -, =, >, \text{and}, \text{or}\}$
<b>not</b> $p$	
Command $\ni c ::= \bar{x} := \bar{p}$	assignment
$c; c$	sequential composition
<b>return</b> $\bar{q}$	function return
<b>assume</b> $q$	assumption
<b>assert</b> $q$	assertion

Let  $t_i$  denote the  $i$ -th element in the sequence  $\bar{t}$ . The **nondet** expression evaluates to a type-safe nondeterministic value. Note that simultaneous assignments are allowed in our language. To execute a simultaneous assignment, all expressions on the right hand side are first evaluated and then assign to respective variables. The **return** command accepts several expressions as arguments. Together with simultaneous assignments, functions can have several return values.

A function  $\mathbf{f}$  is represented as a *control flow graph (CFG)*  $G^{\mathbf{f}} = \langle V, E, \text{cmd}^{\mathbf{f}}, \bar{\mathbf{u}}^{\mathbf{f}}, \bar{\mathbf{r}}^{\mathbf{f}}, s, e \rangle$  where the nodes in  $V$  are *program locations*,  $E \subseteq V \times V$  are *edges*, each edge  $(\ell, \ell') \in E$  is labeled by the command  $\text{cmd}^{\mathbf{f}}(\ell, \ell')$ ,  $\bar{\mathbf{u}}^{\mathbf{f}}$  and  $\bar{\mathbf{r}}^{\mathbf{f}}$  are *formal parameters* and *return variables* of  $\mathbf{f}$ , and  $s, e \in V$  are the *entry* and *exit* locations of  $\mathbf{f}$ . The superscript in  $G^{\mathbf{f}}$  denotes the CFG corresponds to the function  $\mathbf{f}$ . The special **main** function specifies where a program starts. To simplify presentation, we assume the functions in a program use disjoint sets of variables and the values of formal parameters never change in a function. Notice that this will not affect the expressiveness of a CFG because one can still make copies of formal parameters by assignments and change the values of the copied versions. Also we assume that there are no global variables because they can be simulated by allowing simultaneous assignment to return variables.

Figure 1 shows control flow graphs for the McCarthy 91 program. The **main** function assumes the variable  $\mathbf{n}$  is non-negative. It then checks if the result of  $\text{mc91}(\mathbf{n})$  is no less than 90 (Figure 1a). The  $\text{mc91}$  function branches on whether the variable  $\mathbf{m}$  is greater than 100. If so, it returns  $\mathbf{m} - 10$ . Otherwise,  $\text{mc91}(\mathbf{m})$  stores the result of  $\text{mc91}(\mathbf{m} + 11)$  in  $\mathbf{s}$ , and returns the result of  $\text{mc91}(\mathbf{s})$  (Figure 1b). Observe that the **assume** command models a conditional branch in the figure. Loops can be modeled similarly.

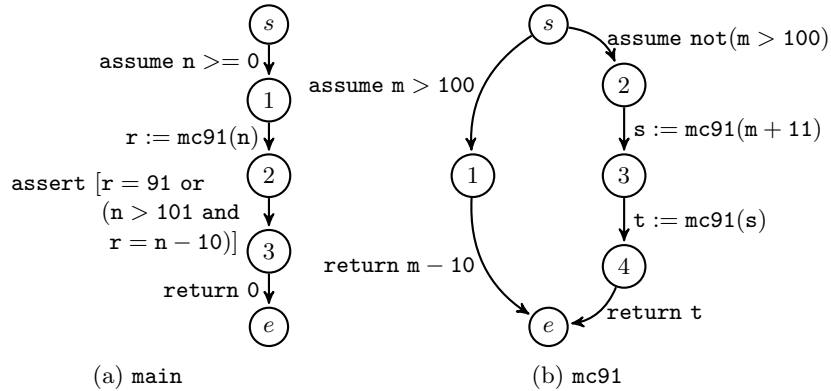


Fig. 1: McCarthy 91

Let  $G^{\mathbf{f}} = \langle V, E, \text{cmd}^{\mathbf{f}}, \bar{\mathbf{u}}^{\mathbf{f}}, \bar{\mathbf{r}}^{\mathbf{f}}, s, e \rangle$  be a CFG. An *inductive invariant*  $\Pi(G^{\mathbf{f}}, I_0) = \{I_\ell : \ell \in V\}$  for  $G^{\mathbf{f}}$  from  $I_0$  is a set of first-order logic formu-

lae such that  $I_s = I_0$ , and for every  $(\ell, \ell') \in E$

$$I_\ell \wedge \tau_{\text{cmd}(\ell, \ell')} \implies I_{\ell'}$$

where  $I'$  is obtained by replacing every  $\mathbf{x} \in \mathbf{Vars}$  in  $I$  with  $\mathbf{x}' \in \mathbf{Vars}'$ , and  $\tau_{\text{cmd}(\ell, \ell')}$  specifies the semantics of the command  $\text{cmd}(\ell, \ell')$ . An inductive invariant  $\Pi(G^f, I_0)$  is an over-approximation to the computation of  $G^f$  from  $I_0$ . More precisely, assume that the function  $\mathbf{f}$  starts from a state satisfying  $I_0$ . For every  $\ell \in V$ ,  $G^f$  must arrive in a state satisfying  $I_\ell$  when the computation reaches  $\ell$ .

Let  $T$  be a program fragment (it can be either a function represented as a CFG or a sequence of program commands).  $P$  and  $Q$  are logic formulae. A *Hoare triple*  $\langle P \rangle T \langle Q \rangle$  specifies that the program fragment  $T$  will reach a program state satisfying  $Q$  provided that  $T$  starts with a program state satisfying  $P$  and terminates. The formula  $P$  is called the *precondition* and  $Q$  is the *postcondition* of the Hoare triple. We use the standard proof rules for weak correctness with two additional rules for the assumption and assertion commands:

$$\text{Assume} \frac{}{\langle P \rangle \text{ assume } q \langle P \wedge q \rangle} \quad \text{Assert} \frac{P \implies q}{\langle P \rangle \text{ assert } q \langle P \rangle}$$

The **assume** command excludes all computation not satisfying the given expression. The **assert** command terminates computation abnormally if the given expression is not implied by the precondition. No postcondition can be guaranteed in such a case. Observe that an inductive invariant  $\Pi(G^f, I_0)$  establishes  $\langle I_0 \rangle G^f \langle I_e \rangle$ . A *program analyzer* accepts programs as inputs and checks if all assertions (specified by the **assert** command) are satisfied. One way to implement program analyzers is to compute inductive invariants.

**Proposition 1.** *Let  $G^f = \langle V, E, \text{cmd}^f, \bar{\mathbf{u}}^f, \bar{\mathbf{r}}^f, s, e \rangle$  be a CFG and  $\Pi(G^f, \text{true})$  be an inductive invariant for  $G^f$  from **true**. If  $\models I_\ell \implies B_\ell$  for every edge  $(\ell, \ell') \in E$  with  $\text{cmd}(\ell, \ell') = \text{assert}(B_\ell)$ , then all assertions in  $G^f$  are satisfied.*

A program analyzer checks assertions by computing inductive invariants is called an *inductive* program analyzer. Note that an inductive program analyzer need not give any information when an assertion fails. Indeed, most inductive program analyzers simply report false positives when inductive invariants are too coarse. A *program checker* is a program analyzer that returns an error trace when an assertion fails; an *error trace* is a sequence of variable valuations from the program entry to the failed assertion. Rather than reporting false positives, program checkers have to return error traces to witness failed assertions. A *recursion-free inductive program analyzer* is a program analyzer that checks recursion-free programs by computing inductive invariants, and reports an error trace when an assertion fails. Several recursion-free inductive program analyzer are available such as CPACHECKER, BLAST, UFO, ASTRÉE. Our goal is to check recursive programs by using a recursion-free inductive program analyzer as a black box.

### 3 Overview

Let BASICANALYZER denote a recursion-free inductive program analyzer and a program  $P = \{G^{\text{main}}\} \cup \{G^f : \mathbf{f} \text{ is a function}\}$  consists of the CFGs of the

**main** function and functions that may be invoked (transitively) from **main**. Since non-recursive functions can be replaced by their control flow graphs after proper variable renaming, we assume that  $P$  only contains the **main** and recursive functions. If  $P$  does not contain recursive functions, BASICANALYZER is able to check  $P$  by computing inductive invariants.

When  $P$  contains recursive functions, we transform  $G^{\text{main}}$  into a recursion-free program  $\underline{G}^{\text{main}}$ . The program  $\underline{G}^{\text{main}}$  under-approximates the computation of  $G^{\text{main}}$ . That is, every computation of  $\underline{G}^{\text{main}}$  is also a computation of  $G^{\text{main}}$ . If BASICANALYZER finds an error in  $\underline{G}^{\text{main}}$ , our algorithm terminates and reports it. Otherwise, BASICANALYZER has computed an inductive invariant for the recursive-free under-approximation  $\underline{G}^{\text{main}}$ . Our algorithm updates function summaries of functions in  $P$  from the inductive invariant of  $\underline{G}^{\text{main}}$ . It then checks if every function summary over-approximates the computation of the corresponding function. If so, the algorithm terminates and reports that all assertions in  $P$  are satisfied. If a function summary does not over-approximate the computation, our algorithm unwinds the recursive function and reiterates (Algorithm 1).

```

Input: A program  $P = \{G^{\text{main}}\} \cup \{G^f : f \text{ is a function}\}$ 
 $k \leftarrow 0$ ;
 $P_0 \leftarrow P$ ;
repeat
   $k \leftarrow k + 1$ ;
   $P_k \leftarrow$  unwind every CFGs in  $P_{k-1}$ ;
  switch BASICANALYZER( $\underline{G}_k^{\text{main}}$ ) do
    case  $\text{Pass}(\Pi(\underline{G}_k^{\text{main}}, \text{true}))$ :  $S := \text{UpdateSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$ ;
    case Error: return Error;
   $\text{complete?} \leftarrow \text{CheckSummary}(P_k, S)$ ;
until  $\text{complete?}$ ;
return  $\text{Pass}(\Pi(\underline{G}_k^{\text{main}}, \text{true})), S$ ;

```

**Algorithm 1:** Overview

To see how to under-approximate computation, consider a control flow graph  $G_k^{\text{main}}$ . The under-approximation  $\underline{G}_k^{\text{main}}$  is obtained by substituting the command **assume false** for every command with recursive function calls (Figure 2). The substitution effectively blocks all recursive invocations. Any computation of  $\underline{G}_k^{\text{main}}$  hence is also a computation of  $G_k^{\text{main}}$ . Note that  $\underline{G}_k^{\text{main}}$  is recursion-free. BASICANALYZER is able to check the under-approximation  $\underline{G}_k^{\text{main}}$ .

When BASICANALYZER does not find any error in the under-approximation  $\underline{G}_k^{\text{main}}$ , it computes an inductive invariant  $\Pi(\underline{G}_k^{\text{main}}, \text{true})$ . Our algorithm then updates summaries of functions in  $P$ . For each function  $f$  with formal parameters  $\bar{u}^f$  and return variables  $\bar{r}^f$ , a *function summary* for  $f$  is a first-order conjunctive formula which specifies the relation between its formal parameters and return variables. The algorithm  $\text{UpdateSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$  updates function summaries  $S$  by inspecting the inductive invariant  $\Pi(\underline{G}_k^{\text{main}}, \text{true})$  (Section 4.3).

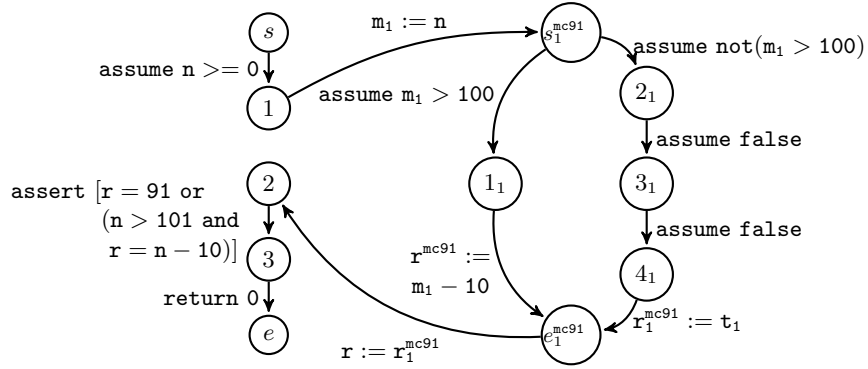


Fig. 2: Under Approximation of McCarthy 91

After function summaries are updated, Algorithm 1 verifies whether function summaries correctly specify computations of functions by invoking  $\text{CheckSummary}(P_k, S)$ . The algorithm  $\text{CheckSummary}(P_k, S)$  checks this by constructing a control flow graph  $\tilde{G}^f$  with additional assertions for each function  $f$  and verifying  $\tilde{G}^f$  with  $\text{BASICANALYZER}$ . The control flow graph  $\tilde{G}^f$  is obtained by substituting function summaries for function calls. We transform  $G^f$  into  $\tilde{G}^f$  by the following three steps:

1. Replace every function call by instantiating the summary for the callee;
2. Replace every return command by assignments to return variables;
3. Add an assertion to validate the summary at the end.

Figure 3 shows the control flow graph  $\tilde{G}^{\text{mc91}}$  with the function summary  $S[\text{mc91}] = \text{not}(m \geq 0)$ . Observe that  $\tilde{G}^{\text{mc91}}$  is recursion-free.  $\text{BASICANALYZER}$  is able to check  $\tilde{G}^{\text{mc91}}$ .

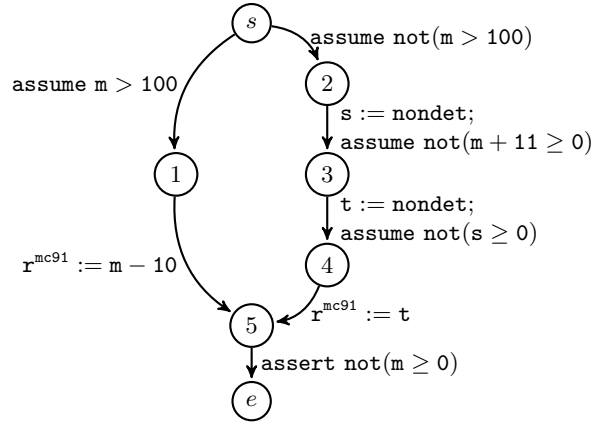


Fig. 3: Check Summary in McCarthy 91

In order to refine approximations, our algorithm unwinds recursive functions as usual. More precisely, consider a recursive function  $\mathbf{f}$  with formal parameters  $\bar{\mathbf{u}}^{\mathbf{f}}$  and return variables  $\bar{\mathbf{r}}^{\mathbf{f}}$ . Let  $G^{\mathbf{f}}$  be the control flow graph of  $\mathbf{f}$  and  $G_k^{\text{main}}$  be a control flow graph that invokes  $\mathbf{f}$ . To unwind  $\mathbf{f}$  in  $G_k^{\text{main}}$ , we first construct a control flow graph  $H^{\mathbf{f}}$  by replacing every `return  $\bar{q}$`  command in  $\mathbf{f}$  with the assignment  $\bar{\mathbf{r}}^{\mathbf{f}} := \bar{q}$ . For each edge  $(\ell, \ell')$  labeled with the command  $\bar{\mathbf{x}} := \mathbf{f}(\bar{\mathbf{p}})$  in  $G_k^{\text{main}}$ , we remove the edge  $(\ell, \ell')$ , make a fresh  $K^{\mathbf{f}}$  by renaming all nodes and variables in  $H^{\mathbf{f}}$ , add an edge from  $\ell$  to the entry node of  $K^{\mathbf{f}}$  that assigns  $\bar{\mathbf{p}}$  to fresh copies of formal parameters in  $K^{\mathbf{f}}$ , and another edge from the exit node to  $\ell'$  that assigns fresh copies of return variables to  $\bar{\mathbf{x}}$ . The control flow graph  $G_{k+1}^{\text{main}}$  is obtained by unwinding every function call in  $G_k^{\text{main}}$ . Figure 4 shows the control flow graph obtained by unwinding `main` twice. Note that the unwinding graph  $G_{k+1}^{\text{main}}$  still has recursive function calls. Its under-approximation  $\underline{G}_{k+1}^{\text{main}}$  is more accurate than  $\underline{G}_k^{\text{main}}$ .

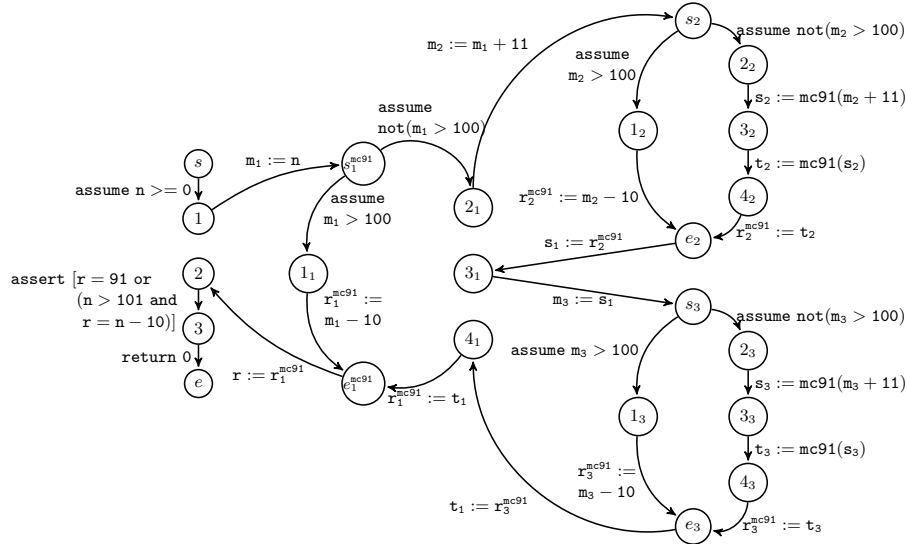


Fig. 4: Unwinding McCarthy 91

## 4 Proving via Transformation

We give details of the constructions and establish the soundness of Algorithm 1. Our goal is to establish the following theorem:

**Theorem 1.** *Let  $G^{\text{main}} = \langle V, E, \text{cmd}^{\text{main}}, \bar{\mathbf{u}}^{\text{main}}, \bar{\mathbf{r}}^{\text{main}}, s, e \rangle$  be a control flow graph in  $P$ . If Algorithm 1 returns *Pass*, there is an inductive invariant  $\Pi(G^{\text{main}}, \text{true})$  such that  $I_\ell \implies B_\ell$  for every  $(\ell, \ell') \in E$  with  $\text{cmd}^{\text{main}}(\ell, \ell') = \text{assert } B_\ell$ .*

By Proposition 1, it follows that all assertions in  $G^{\text{main}}$  are satisfied. Moreover, by the semantics of the proof rules for recursive functions and the `assume` command, all assertions in the program are satisfied.

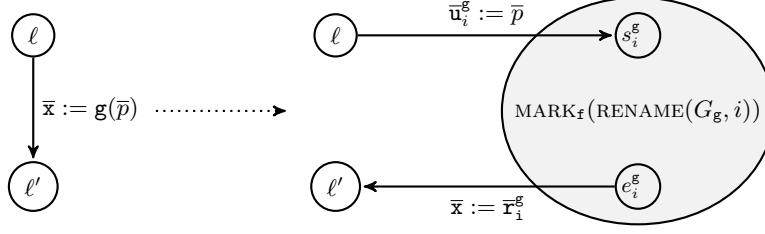


Fig. 5: Unwinding Function Calls

#### 4.1 Unwinding

We first define the rename function  $\text{RENAME}(G^f, i)$ . It returns a CFG  $\langle V_i, E_i, \text{cmd}_i^f, \bar{u}_i^f, \bar{r}_i^f, s_i, e_i \rangle$  obtained by first replacing every return command **return**  $\bar{q}$  by assignments to return variables  $\bar{r}^f := \bar{q}$  and then renaming all variables and locations in  $G^f$  with the index value  $i$ . The function  $\text{UNWIND}(G^f)$  returns a CFG  $K^f$  obtained by replacing all function call edges in  $G^f$  with the CFG of the called function after renaming. In order to help extracting summaries from the  $K^f$ ,  $\text{UNWIND}(G^f)$  annotates in  $K^f$  the outermost pair of the entry and exit locations  $s_i$  and  $e_i$  of each unwound functions  $g$  with an additional superscript  $g$ , i.e.,  $s_i^g$  and  $e_i^g$ . The formal definition is given below.

Given a CFG  $G^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$ . We use  $\hat{E} = \{e \in E : \text{cmd}^f(e) = (\bar{x} := g(\bar{p}))\}$  to denote the set of function call edges in  $E$  and define a function  $\text{IDX}(e)$  that maps a call edge  $e$  to a unique index value. The function  $\text{MARK}_f(G^g)$  returns a CFG that is identical to  $G^g$ , except that for the case that no location with superscript  $g$  appears in  $V$  (the locations of  $G^f$ ), it annotates the entry and exit locations,  $s_k$  and  $e_k$ , of the returned CFG with superscript  $g$ , i.e.,  $s_k^g$  and  $e_k^g$ . Note that for each unwinding of function call we mark only the outermost pair of its entry and exit locations. Formally,  $\text{UNWIND}(G^f) = \langle V_u, E_u, \text{cmd}_u^f, \bar{u}_u^f, \bar{r}_u^f, s, e \rangle$  such that (1)  $V_u = V \cup \bigcup \{V_i : (\ell, \ell') \in \hat{E} \wedge \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \wedge \text{IDX}(\ell, \ell') = i \wedge \text{MARK}_f(\text{RENAME}(G^g, i)) = \langle V_i, E_i, \text{cmd}_i^g, \bar{u}_i^g, \bar{r}_i^g, s', e' \rangle\}$  (2)  $E_u = E \setminus \hat{E} \cup \bigcup \{E_i \cup \{(\ell, s'), (e', \ell')\} : (\ell, \ell') \in \hat{E} \wedge \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \wedge \text{IDX}(\ell, \ell') = i \wedge \text{MARK}_f(\text{RENAME}(G^g, i)) = \langle V_i, E_i, \text{cmd}_i^g, \bar{u}_i^g, \bar{r}_i^g, s', e' \rangle\}$  with  $\text{cmd}_u^f(\ell, s') = (\bar{u}_i^g := \bar{p})$  and  $\text{cmd}_u^f(e', \ell') = (\bar{x} := \bar{r}_i^g)$ .

**Proposition 2.** Let  $G^f$  be a control flow graph.  $P$  and  $Q$  are logic formulae with free variables over program variables of  $G^f$ .  $\langle P \rangle G^f \langle Q \rangle$  if and only if  $\langle P \rangle \text{UNWIND}(G^f) \langle Q \rangle$ .

Essentially,  $G^f$  and  $\text{UNWIND}(G^f)$  represent the same function  $f$ . The only difference is that the latter has more program variables after unwinding, but this does not affect the states over program variables of  $G^f$  before and after the function.

#### 4.2 Under Approximation

Let  $G^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  be a control flow graph. The control flow graph  $\underline{G}^f = \langle V, E, \underline{\text{cmd}}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  is obtained by replacing every function calls in  $G$



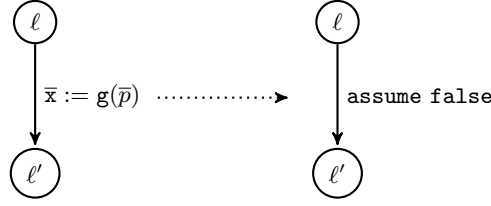


Fig. 6: Under Approximation

with **assume false**. That is,

$$\underline{\text{cmd}}^f(\ell, \ell') = \begin{cases} \text{cmd}^f(\ell, \ell') & \text{if } \text{cmd}^f(\ell, \ell') \text{ does not contain function calls} \\ \text{assume false} & \text{otherwise} \end{cases}$$

**Proposition 3.** *Let  $G^f$  be a control flow graph.  $P$  and  $Q$  are logic formulae with free variables over program variables of  $G^f$ . If  $\langle P \rangle G^f \langle Q \rangle$ , then  $\langle P \rangle \underline{G}^f \langle Q \rangle$ .*

The above holds because the computation of  $\underline{G}^f$  under-approximates the computation of  $G^f$ . If all computation of  $G^f$  from a state satisfying  $P$  always ends with a state satisfying  $Q$ , the same should also hold for the computation of  $\underline{G}^f$ .

### 4.3 Updating Summary

Let the CFG for the **main** function  $\underline{G}_k^{\text{main}} = \langle V, E, \underline{\text{cmd}}^{\text{main}}, \underline{\mathbf{u}}^{\text{main}}, \underline{\mathbf{r}}^{\text{main}}, s, e \rangle$ . Function  $\text{UpdateSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$  extracts summaries from the inductive invariant  $\Pi(\underline{G}_k^{\text{main}}, \text{true}) = \{I_\ell : \ell \in V\}$  (Algorithm 2).

**Input:**  $P$ : a program;  $\{I_\ell : \ell \in V\}$ : an inductive invariant of  $\underline{G}_k^{\text{main}}$   
**Output:**  $S[\bullet]$ : function summaries  
**foreach** function  $\mathbf{f}$  in the program  $P$  **do**  
     $S[\mathbf{f}] := \text{true};$   
    **foreach** pair of locations  $(s_i^f, e_i^f) \in V \times V$  **do**  
        **if**  $I_{s_i^f}$  does not contain any return variable of  $\mathbf{f}$  **then**  
             $S[\mathbf{f}] := S[\mathbf{f}] \wedge \forall X_{\mathbf{f}}. (I_{s_i^f} \implies I_{e_i^f});$   
        **else**  $S[\mathbf{f}] := S[\mathbf{f}] \wedge \forall X_{\mathbf{f}}. I_{e_i^f};$   
**return**  $S[\bullet];$   
**Algorithm 2:**  $\text{UpdateSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$

For each function  $\mathbf{f}$  in the program  $P$ , we first initialize its summary  $S[\mathbf{f}]$  to **true**. The set  $X_{\mathbf{f}}$  contains all variables appearing in  $\underline{G}_k^{\text{main}}$  except the set of formal parameter and return variables of  $\mathbf{f}$ . For each pairs of locations  $(s_i^f, e_i^f) \in V \times V$  in  $\underline{G}_k^{\text{main}}$ , if the invariant of location  $s_i^f$  does not contain any return variable of  $\mathbf{f}$ , we update  $S[\mathbf{f}]$  to the formula  $S[\mathbf{f}] \wedge \forall X_{\mathbf{f}}. (I_{s_i^f} \implies I_{e_i^f})$ . Otherwise, we update it to a more restricted version  $S[\mathbf{f}] \wedge \forall X_{\mathbf{f}}. I_{e_i^f}$ .

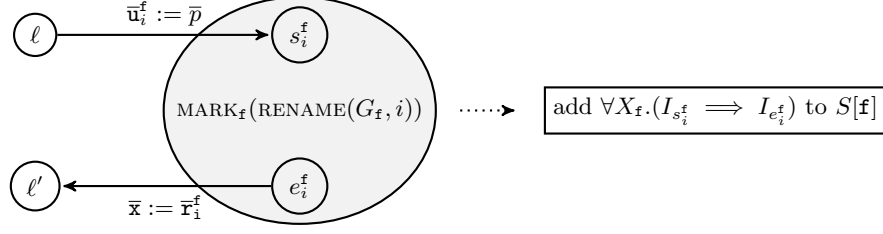


Fig. 7: Updating Summary

**Proposition 4.** *Let  $Q$  be a formula over all variables in  $\underline{G}_k^{\text{main}}$  except  $\bar{r}^f$ . We have  $\langle Q \rangle \bar{r}^f := f(\bar{u}^f) \langle Q \rangle$ .*

The proposition holds because the only possible overlap of variables in  $Q$  and in  $\bar{r}^f := f(\bar{u}^f)$  are the formal parameters  $\bar{u}^f$ . However, we assume that values of formal parameters do not change in a function (see Section 2), hence the values of all variables in  $Q$  stay the same after the execution of the function call  $\bar{r}^f := f(\bar{u}^f)$ .

**Proposition 5.** *Given the CFG  $\underline{G}_k^{\text{main}} = \langle V, E, \text{cmd}^{\text{main}}, \bar{u}^{\text{main}}, \bar{r}^{\text{main}}, s, e \rangle$ . If  $\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle S[f] \rangle$  holds, then  $\langle I_{s_i^f} \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{e_i^f} \rangle$  for all  $(s_i^f, e_i^f) \in V \times V$ .*

For each pair  $(s_i^f, e_i^f) \in V \times V$ , we consider two cases:

1.  $I_{s_i^f}$  contains some return variables of  $f$ :

In this case, the conjunct  $\forall X_f. I_{e_i^f}$  is a part of  $S[f]$ , we then have

$$\frac{\frac{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle S[f] \rangle}{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle \forall X_f. I_{e_i^f} \rangle} \text{Weakening}}{\frac{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{e_i^f} \rangle}{\langle I_{s_i^f} \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{e_i^f} \rangle} \text{Strengthening}} \text{Weakening}$$

2.  $I_{s_i^f}$  does not contain any return variables of  $f$ :

In this case, the conjunct  $\forall X_f. (I_{s_i^f} \implies I_{e_i^f})$  is a part of  $S[f]$ , we then have

$$\text{Prop. 4} \frac{\frac{\frac{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle S[f] \rangle}{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle \forall X_f. (I_{s_i^f} \implies I_{e_i^f}) \rangle}}{\frac{\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{s_i^f} \implies I_{e_i^f} \rangle}{\langle I_{s_k^f}^- \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{s_k^f}^- \rangle}}}{\frac{\langle I_{s_k^f}^- \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{e_k^f}^- \rangle}{\langle I_{s_k^f}^- \rangle \bar{r}^f := f(\bar{u}^f) \langle I_{e_k^f}^- \rangle}} \text{Strengthening}$$

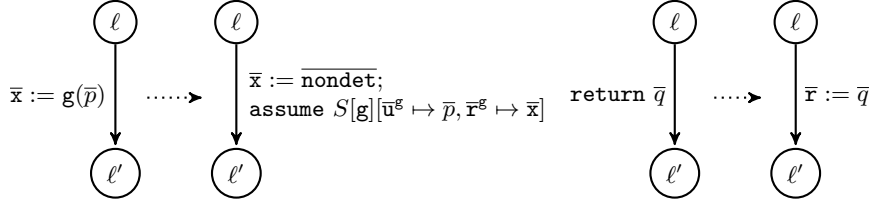


Fig. 8: Checking Summary

#### 4.4 Checking Summary

Here we explain how to handle the function  $\text{CheckSummary}(P_k, S[\bullet])$ , where  $P_k$  is an unwound program and  $S[\bullet]$  is an array of function summaries. Let  $G_k^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  be a control flow graph for the function  $f$  in  $P_k$ . In order to check whether the function summary  $S[f]$  for  $f$  specifies the relation between the formal parameters and return values of  $f$ , we define another control flow graph  $\hat{G}_{k,S}^f = \langle V, E, \hat{\text{cmd}}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  where

$$\hat{\text{cmd}}^f(\ell, \ell') = \begin{cases} \bar{x} := \overline{\text{nondet}}; \text{assume } S[g][\bar{u}^g \mapsto \bar{p}, \bar{r}^g \mapsto \bar{x}] & \text{if } \text{cmd}^f(\ell, \ell') = \bar{x} := g(\bar{p}) \\ \bar{r}^f := \bar{q} & \text{if } \text{cmd}^f(\ell, \ell') = \text{return } \bar{q} \\ \text{cmd}^f(\ell, \ell') & \text{otherwise} \end{cases}$$

The control flow graph  $\hat{G}_{k,S}^f$  replaces every function call in  $G_k^f$  by instantiating a function summary. Using the proof rule for recursive functions, we have the following proposition:

**Proposition 6.** *Let  $G_k^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  be the control flow graph for the function  $f$  and  $S[\bullet]$  be an array of logic formulae over the formal parameters and return variables of each function. If  $\langle \text{true} \rangle \hat{G}_{k,S}^g \langle S[g] \rangle$  for every function  $g$  in  $P$ , then  $\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle S[f] \rangle$ .*

It is easy to check  $\langle \text{true} \rangle \hat{G}_{k,S}^g \langle S[g] \rangle$  by program analysis. Let  $G_k^f$  be the control flow graph for the function  $f$  and  $\hat{G}_{k,S}^g = \langle V, E, \hat{\text{cmd}}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  as above. Consider another control flow graph  $\tilde{G}_{k,S}^f = \langle \tilde{V}, \tilde{E}, \tilde{\text{cmd}}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  where

$$\begin{aligned} \tilde{V} &= V \cup \{\tilde{e}\} \\ \tilde{E} &= E \cup \{(e, \tilde{e})\} \\ \tilde{\text{cmd}}^f(\ell, \ell') &= \begin{cases} \hat{\text{cmd}}^f(\ell, \ell') & \text{if } (\ell, \ell') \in E \\ \text{assert } S[f] & \text{if } (\ell, \ell') = (e, \tilde{e}) \end{cases} \end{aligned}$$

**Corollary 1.** *Let  $G_k^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$  be the control flow graph for the function  $f$  and  $S[\bullet]$  be an array of logic formulae over the formal parameters and return variables of each function. If  $\text{BASICCHECKER}(\tilde{G}_{k,S}^g)$  returns Pass for every function  $g$  in  $P$ , then  $\langle \text{true} \rangle \bar{r}^f := f(\bar{u}^f) \langle S[f] \rangle$ .*

**Input:**  $P$  : an unwound program;  $S[\bullet]$  : an array of function summaries  
**Output:** **true** if all function summaries are valid; **false** otherwise  
**foreach** function  $G_k^g \in P$  **do**  
    | **if** BASICCHECKER( $\tilde{G}_{k,S}^g$ )  $\neq$  *Pass* **then return false;**  
**return true;**

**Algorithm 3:** CheckSummary( $P_k, S$ )

## 4.5 Correctness

We are ready to sketch the proof of Theorem 1. Assume Algorithm 1 returns *Pass*( $\Pi(\underline{G}_k^{\text{main}}, \text{true})$ ) and  $S[\bullet]$  on the input control flow graph  $G^{\text{main}} = \langle V, E, \text{cmd}^{\text{main}}, \bar{u}^{\text{main}}, \bar{r}^{\text{main}}, s, e \rangle$ . Let  $\underline{G}_k^{\text{main}} = \langle V_k, \underline{E}_k, \underline{\text{cmd}}_k^{\text{main}}, \bar{u}^{\text{main}}, \bar{r}^{\text{main}}, s, e \rangle$  and  $\Pi(\underline{G}_k^{\text{main}}, \text{true}) = \{\underline{I}_\ell : \ell \in V_k\}$ . By the definition of inductive invariants, we have  $\langle \underline{I}_\ell \rangle \underline{\text{cmd}}_k^{\text{main}}(\ell, \ell') \langle \underline{I}_{\ell'} \rangle$  for every  $(\ell, \ell') \in \underline{E}_k$ . Moreover,  $V \subseteq V_k$  since  $\underline{G}_k^{\text{main}}$  is obtained by unwinding  $G^{\text{main}}$ . Define  $\Gamma(G^{\text{main}}, \text{true}) = \{\underline{I}_\ell \in \Pi(\underline{G}_k^{\text{main}}, \text{true}) : \ell \in V\}$ . We claim  $\Gamma(G^{\text{main}}, \text{true})$  is in fact an inductive invariant for  $G^{\text{main}}$ .

Let  $\hat{E} = \{(\ell, \ell') \in E : \text{cmd}^{\text{main}}(\ell, \ell') = \bar{x} := f(\bar{p})\}$ . We have  $\text{cmd}^{\text{main}}(\ell, \ell') = \underline{\text{cmd}}_k^{\text{main}}(\ell, \ell')$  for every  $(\ell, \ell') \in E \setminus \hat{E}$ . Thus  $\langle \underline{I}_\ell \rangle \text{cmd}^{\text{main}}(\ell, \ell') \langle \underline{I}_{\ell'} \rangle$  for every  $(\ell, \ell') \in E \setminus \hat{E}$  by the definition of  $\Gamma(G, \text{true})$  and the inductiveness of  $\Pi(\underline{G}_k^{\text{main}}, \text{true})$ . It suffices to show that

$$\langle \underline{I}_\ell \rangle \bar{x} := f(\bar{p}) \langle \underline{I}_{\ell'} \rangle \text{ or, equivalently, } \langle \underline{I}_\ell \rangle \bar{u}^f := \bar{p}; \bar{r}^f := f(\bar{u}^f); \bar{x} := \bar{r}^f \langle \underline{I}_{\ell'} \rangle$$

for every  $(\ell, \ell') \in \hat{E}$ . By the inductiveness of  $\Pi(\underline{G}_k^{\text{main}}, \text{true})$ , we have  $\langle \underline{I}_\ell \rangle \bar{u}^f := \bar{p} \langle \underline{I}_{s_k^f} \rangle$  and  $\langle \underline{I}_{e_k^f} \rangle \bar{x} := \bar{r}^f \langle \underline{I}_{\ell'} \rangle$ . Moreover,  $\langle \underline{I}_{s_k^f} \rangle \bar{r}^f := f(\bar{u}^f) \langle \underline{I}_{e_k^f} \rangle$  by Proposition 5 and 6. Therefore

$$\frac{\langle \underline{I}_\ell \rangle \bar{u}^f := \bar{p} \langle \underline{I}_{s_k^f} \rangle \quad \langle \underline{I}_{s_k^f} \rangle \bar{r}^f := f(\bar{u}^f) \langle \underline{I}_{e_k^f} \rangle \quad \langle \underline{I}_{e_k^f} \rangle \bar{x} := \bar{r}^f \langle \underline{I}_{\ell'} \rangle}{\langle \underline{I}_\ell \rangle \bar{u}^f := \bar{p}; \bar{r}^f := f(\bar{u}^f); \bar{x} := \bar{r}^f \langle \underline{I}_{\ell'} \rangle} \quad \langle \underline{I}_\ell \rangle \bar{x} := f(\bar{p}) \langle \underline{I}_{\ell'} \rangle$$

## 5 Experiments

A prototype tool of our approach has been implemented with CPACHECKER 1.2.11-svcomp14b<sup>3</sup> as the underlying intraprocedural analyzer. In addition, because CPACHECKER does not support universal quantifiers in the expression of an **assume** command, we used Redlog [4] for quantifier elimination. To evaluate our tool, we performed experiments with the benchmarks from the **recursive** category in the 2014 Competition on Software Verification (SV-COMP 2014) [1]

<sup>3</sup> We use script/cpa.sh to invoke CPACHECKER and use the configuration file available at <https://github.com/fmlab-iis/transformer/blob/master/tool/verifier-conf/myCPA-PredAbstract-LIA.properties>.

and followed the rules and the score schema (shown in Table 1) of the competition. The experimental results show that our tool is quite competitive even compared with the winners of the competition. It is a solid evidence that our approach not merely extend program analyzer to handle recursion but also provide comparable effectiveness.

In the experiments, our tool was compared with four participants of SV-COMP 2014, namely BLAST 2.7.2<sup>4</sup> [9], CBMC 4.5-sv-comp-2014 [12] with a wrapper `cbmc-wrapper.sh`<sup>5</sup>, ULTIMATE AUTOMIZER [15], and ULTIMATE KOJAK [5]. The latter three tools are the top three winners of the **recursive** category in SV-COMP 2014. The recursive programs from the benchmarks of the **recursive** category include 16 bug-free and 7 buggy C programs. The experiments were performed on a virtual machine with 4 GB of memory running 64-bit Ubuntu 12.04 LTS. The virtual machine ran on a host with an Intel Core i7-870 Quad-Core CPU running 64-bit Windows 7. The timeout of a verification task is 900 seconds.

The experimental results are summarized in Table 2 where  $k$  is the number of unwindings of recursive functions in Algorithm 1, Time is measured in seconds, the superscript ! or ? indicates that the returned result is respectively incorrect or unknown, E indicates exceptions, and T.O. indicates timeouts. The parenthesized numbers of CBMC are obtained by excluding certain cases, which will be explained later.

The results show that CBMC outperforms all the other tools. However, CBMC reports safe if no bug is found in a program within a given time bound<sup>6</sup>, which is set to 850 seconds in `cbmc-wrapper.sh`. In this case, the behaviors of the program within certain length bounds are proven to be safe, but the absence of bugs is not guaranteed (see `Addition03.false.c` in Table 2 for a counterexample). If we ignore such cases in the experiments, CBMC will obtain a score of 14, and the gap between the scores of CBMC and our tool becomes much smaller. Moreover, this gap may be narrowed if we turn on some important optimizations such as adjustment of block encoding provided in CPACHECKER. We chose to disable the optimizations in order to simplify the implementation of our prototype tool.

Compared to ULTIMATE AUTOMIZER, ULTIMATE KOJAK, and BLAST, our tool can verify more programs and obtain a higher score. The scores of our tool and ULTIMATE AUTOMIZER are very close mainly because of a false positive produced by our tool. The false positive in fact came from a spurious error trace reported by CPACHECKER because modulo operation is approximated in CPACHECKER. If this case is excluded, our tool can obtain a score of 16.

Besides the four participants of SV-COMP 2014, we also tried to compare our tool with WHALE. Unfortunately, we always got segmentation fault when running WHALE on the recursive programs in Table 2.

<sup>4</sup> We use the arguments `-alias empty -enable-recursion -nopprofile -cref -sv-comp -lattice -include-lattice symb -nosserr` with BLAST.

<sup>5</sup> The wrapper `cbmc-wrapper.sh` is provided by CBMC 4.5-sv-comp-2014, which is a special version for SV-COMP 2014.

<sup>6</sup> This was confirmed in a private communication with the developers of CBMC.

Table 1: Score schema in SV-COMP 2014.

Points	Program Correctness	Reported Result
0	TRUE or FALSE	UNKNOWN (due to timeout or exceptions)
+1	FALSE	FALSE
-4	TRUE	FALSE
+2	TRUE	TRUE
-8	FALSE	TRUE

Table 2: Experimental results of verifying programs in the **recursive** category of the 2014 Competition on Software Verification.

Program	Our Tool		ULTIMATE AUTOMIZER	ULTIMATE KOJAK	CBMC 4.5	BLAST 2.7.2
	<i>k</i>	Time	Time	Time	Time	Time
Ackermann01_true.c	1	6.466	T.O.	T.O.	850	E
Ackermann02_false.c	4	57.308	4.151	T.O.	1.036	E
Ackermann03_true.c		T.O.	T.O.	T.O.	850	E
Ackermann04_true.c		T.O.	T.O.	T.O.	850	E
Addition01_true.c	2	14.1	T.O.	T.O.	850	E
Addition02_false.c	2	9.937	3.745	3.523	0.26	3.97
Addition03_false.c		T.O.	T.O.	T.O.	850 <sup>†</sup>	E
EvenOdd01_true.c	1	2.882 <sup>†</sup>	T.O.	T.O.	1.293	0.08 <sup>†</sup>
EvenOdd03_false.c	1	2.854	3.163	3.196	0.116	0.08
Fibonacci01_true.c	6	348.406	T.O.	T.O.	850	E
Fibonacci02_true.c		T.O.	60.713	72.138 <sup>?</sup>	0.834	E
Fibonacci03_true.c		T.O.	T.O.	T.O.	850	E
Fibonacci04_false.c	5	107.303	7.438	8.155	0.409	E
Fibonacci05_false.c		T.O.	128.911	23.174	557.18	E
gcd01_true.c	1	6.645	5.369	7.26	850	16.074 <sup>†</sup>
gcd02_true.c		T.O.	T.O.	T.O.	850	E
McCarthy91_false.c	1	2.847	3.152	3.09	0.258	0.086
McCarthy91_true.c	2	12.473	81.32	6.783	850	16.237 <sup>†</sup>
MultCommutative_true.c		T.O.	T.O.	T.O.	850	E
Primes_true.c		T.O.	T.O.	T.O.	850	E
recHanoi01_true.c		T.O.	T.O.	T.O.	850	E
recHanoi02_true.c	1	5.578	T.O.	T.O.	0.749	1.897 <sup>†</sup>
recHanoi03_true.c		T.O.	T.O.	T.O.	0.713	E
correct results	11		9	7	22 (10)	3
false negative	0		0	0	1 (0)	0
false positive	1		0	0	0 (0)	4
score	13		12	9	30 (14)	-13

## 6 Related Works

In [16, 17], a reduction technique for checking context-bounded concurrent programs to sequential analysis is developed. Numerous intraprocedural analysis techniques have been developed over the years. Many tools are in fact freely available (see, for instance, BLAST [9], CPACHECKER [10], and UFO [7]). Interprocedural analysis techniques are also available (see [19, 8, 13, 14, 2, 3] for a hopelessly incomplete list). Recently, recursive analysis attracts new attention. The Competition on Software Verification adds a new category for recursive programs in 2014 [1]. Among the participants, CBMC [12], ULTIMATE AUTOMIZER [15], and ULTIMATE KOJAK [5] are the top three tools for the recursive category. Our work is inspired by WHALE [6]. Similar to WHALE, we apply a Hoare logic proof rule for recursive calls. However, our technique works on control flow graphs and builds on an intraprocedural analysis tool. It is hence very lightweight and modular. Better intraprocedural analysis tools easily give better recursive analysis through our technique. WHALE, on the other hand, analyzes by exploring abstract reachability graphs. Since WHALE extends summary computation and covering relations for recursion, its implementation is more involved. Although WHALE is able to analyze recursive program in theory, its implementation does not appear to support this feature.

## 7 Discussion

The number of unwindings is perhaps the most important factor in our recursive analysis technique (Table 2). We find that CPACHECKER performs poorly when many unwindings are needed. We however do not enable the more efficient block encoding in CPACHECKER for the ease of implementation. One can improve the performance of our algorithm with the efficient but complicated block encoding. A bounded analyzer may also speed up the verification of bounded properties.

Our algorithm extracts function summaries from inductive invariants. There are certainly many heuristics to optimize the computation of function summaries. For instance, some program analyzers return error traces when properties fail. Particularly, a valuation of formal parameters is obtained when CheckSummary (Algorithm 3) returns **false**. If the valuation is not possible in the **main** function, one can use its inductive invariant to refine function summaries. We in fact exploit error traces computed by CPACHECKER in the implementation.

## References

1. Competition on software verification. <http://sv-comp.sosy-lab.org/2014>.
2. Coverity. <http://www.coverity.com/>.
3. Polyspace. <http://www.mathworks.com/products/polyspace/>.
4. Redlog. <http://www.redlog.eu/>.
5. Ultimate kojak. <http://ultimate.informatik.uni-freiburg.de/kojak/>.
6. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

7. Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, pages 672–678, 2012.
8. Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.
9. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5-6):505–525, 2007.
10. Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
11. Edmund M. Clarke, Himanshu Jain, and Nishant Sinha. Grand challenge: Model check software. In *VISSAS*, pages 55–68, 2005.
12. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
13. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astree analyzer. In *ESOP*, pages 21–30, 2005.
14. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012.
15. Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol - (competition contribution). In *TACAS*, pages 641–643, 2013.
16. Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2008.
17. Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.
18. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN 978-3-540-65410-0.
19. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.