Verifying Recursive Programs using Intraprocedural Analyzers

Yu-Fang Chen[1], Chiao Hsieh[1,2], Ming-Hsien Tsai[1], Bow-Yaw Wang[1], and Farn Wang[2]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] 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 complicated program features such as concurrent execution, pointers, recursive function calls, and unbounded basic data types [7]. Subsequently, it is extremely tedious 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 (recursion-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 then transforms the original recursive program into more non-recursive programs with purported function summaries. It finally 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 here, incorporating recursive analysis with existing program analyzers thus only requires minimal implementation efforts. Recursive analysis hence benefits from future advanced intraprocedural analysis with little cost through our lightweight and modular technique.

We implement a prototype using CPAChecker (over 140 thousand lines of Java code) as the underlying program analyzer [6]. In our prototype, 1256 lines of OCaml code are for syntactic transformation and 705 lines of Python code for the rest of the algorithm. 270 lines among them are for extracting function summaries. Since syntactic transformation is independent of underlying program analyzers, only about 14% of code 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 [9].

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

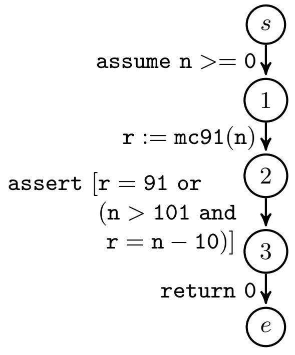
We consider a variant of the WHILE language [17]:

$$
\begin{array}{rcll}
\texttt{Expression} \ni p & ::= & \texttt{x} & \texttt{x} \in \texttt{Vars} \\
 & | & \texttt{false} \mid \texttt{true} \mid \texttt{0} \mid \texttt{1} \mid \ldots & \text{constant} \\
 & | & \texttt{nondet} & \text{nondeterministic value} \\
 & | & \texttt{f}(\overline{p}) & \text{function invocation} \\
 & | & p \odot p & \odot \in \{+, -, =, >, \texttt{and}, \texttt{or}\} \\
 & | & \texttt{not}\ p & \\
\texttt{Command} \ni c & ::= & \overline{\texttt{x}} := \overline{p} & \text{assignment} \\
 & | & c; c & \text{sequential composition} \\
 & | & \texttt{return}\ \overline{p} & \text{function return} \\
 & | & \texttt{assume}\ p & \text{assumption} \\
 & | & \texttt{assert}\ p & \text{assertion}
\end{array}
$$

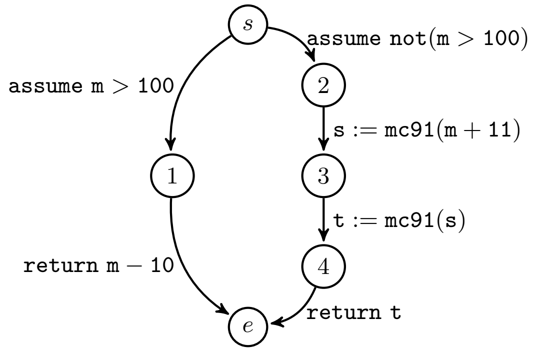
`Vars` denotes the set of *program variables*, and $\texttt{Vars}' = \{\texttt{x}' : \texttt{x} \in \texttt{Vars}\}$ where $\texttt{x}'$ represents the new value of `x` after execution of a command. The `nondet` expression evaluates to a type-safe nondeterministic value. Simultaneous assignments are allowed in our language. To execute a simultaneous assignment, all expressions on the right hand side are first evaluated and then assigned to respective variables. We assume that simultaneous assignments are type-safe in the sense that the number of variables on the left-hand-side always matches that of the right-hand-side. The `return` command accepts several expressions as arguments. Together with simultaneous assignments, functions can have several return values.

A function `f` is represented as a *control flow graph (CFG)* $G^{\texttt{f}} = \langle V, E, \text{cmd}^{\texttt{f}}, \overline{\texttt{u}}^{\texttt{f}}, \overline{\texttt{r}}^{\texttt{f}}, s, e\rangle$ 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 $\text{cmd}^{\texttt{f}}(\ell, \ell')$, $\overline{\texttt{u}}^{\texttt{f}}$ and $\overline{\texttt{r}}^{\texttt{f}}$ are *formal parameters* and *return variables* of `f`, and $s, e \in V$ are the *entry* and *exit* locations of `f`. The superscript in $G^{\texttt{f}}$ denotes the CFG corresponds to the function `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 [3].

Figure 1 shows control flow graphs for the McCarthy 91 program from [16]. The `main` function assumes the variable `n` is non-negative. It then checks if the result of `mc91(n)` is no less than 90 (Figure 1a). The `mc91` function branches on whether the variable `m` is greater than 100. If so, it returns $\texttt{m} - 10$. Otherwise, `mc91(m)` stores the result of `mc91(m + 11)` in `s`, and returns the result of `mc91(s)` (Figure 1b). Observe that a conditional branch is modeled with the `assume` command in the figure. Loops can be modeled similarly.



(a) `main`



(b) `mc91`

Fig. 1: McCarthy 91

Let $G^{\mathtt{f}} = \langle V, E, \mathrm{cmd}^{\mathtt{f}}, \overline{\mathtt{u}}^{\mathtt{f}}, \overline{\mathtt{r}}^{\mathtt{f}}, s, e \rangle$ be a CFG. An *inductive invariant* $\Pi(G^{\mathtt{f}}, I_0) = \{I_\ell : \ell \in V\}$ for $G^{\mathtt{f}}$ from I_0 is a set of first-order logic formulae such that $I_s = I_0$, and for every $(\ell, \ell') \in E$

$$I_\ell \wedge \tau_{\mathrm{cmd}^{\mathtt{f}}(\ell,\ell')} \implies I'_{\ell'}$$

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

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* $(\!|P|\!)T(\!|Q|\!)$ 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 partial correctness with two additional rules for the assumption and assertion commands:

$$\text{Assume}\ \frac{}{(\!|P|\!)\ \mathtt{assume}\ q\ (\!|P \wedge q|\!)} \qquad \text{Assert}\ \frac{P \implies q}{(\!|P|\!)\ \mathtt{assert}\ q\ (\!|P|\!)}$$

The assume command excludes all computation not satisfying the given expression. The assert command aborts the computation if the given expression is not implied by the precondition. No postcondition can be guaranteed in such a case. Observe that an inductive invariant $\Pi(G^{\mathtt{f}}, I_0)$ establishes $(\!|I_0|\!)G^{\mathtt{f}}(\!|I_e|\!)$. 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 $G^{\mathtt{f}} = \langle V, E, cmd^{\mathtt{f}}, \overline{\mathtt{u}}^{\mathtt{f}}, \overline{\mathtt{r}}^{\mathtt{f}}, s, e \rangle$ be a CFG and $\Pi(G^{\mathtt{f}}, \mathtt{true})$ be an inductive invariant for $G^{\mathtt{f}}$ from* true. *If $\models I_\ell \implies B_\ell$ for every edge $(\ell, \ell') \in E$ with $cmd(\ell, \ell') = \mathtt{assert}(B_\ell)$, then all assertions in $G^{\mathtt{f}}$ 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 *recursion-free inductive program analyzer* is a program analyzer that checks recursion-free programs by computing inductive invariants. Several recursion-free inductive program analyzers are available, such as CPAChecker [6], Blast [5], UFO [2], Astrée [10], etc. 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 let a program $P = \{G^{\mathtt{main}}\} \cup \{G^{\mathtt{f}} : \mathtt{f} \text{ is a function}\}$ consist 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 $G^{\texttt{main}}$ into a recursion-free program $\underline{G}^{\texttt{main}}$. The program $\underline{G}^{\texttt{main}}$ under-approximates the computation of $G^{\texttt{main}}$. That is, every computation of $\underline{G}^{\texttt{main}}$ is also a computation of $G^{\texttt{main}}$. If BasicAnalyzer finds an error in $\underline{G}^{\texttt{main}}$, our algorithm terminates and reports it. Otherwise, BasicAnalyzer has computed an inductive invariant for the recursion-free under-approximation $\underline{G}^{\texttt{main}}$. Our algorithm computes function summaries of functions in P from the inductive invariant of $\underline{G}^{\texttt{main}}$. 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 $P = \{G^{\texttt{main}}\} \cup \{G^{\texttt{f}} : \texttt{f} \text{ is a function}\}$
$k \leftarrow 0$;
$P_0 \leftarrow P$;
repeat
    $k \leftarrow k + 1$;
    $P_k \leftarrow$ unwind every CFG in $P_{k-1}$;
    switch BasicAnalyzer $(\underline{G}_k^{\texttt{main}})$ do
        case $Pass(\Pi(\underline{G}_k^{\texttt{main}}, \texttt{true}))$:
            S := ComputeSummary($P_k$, $\Pi(\underline{G}_k^{\texttt{main}}, \texttt{true})$)
        case $Error$: return $Error$ ;
    complete? $\leftarrow$ CheckSummary($P_k, S$);
until $complete?$;
return $Pass(\Pi(\underline{G}_k^{\texttt{main}}, \texttt{true}))$, $S$;
```

Algorithm 1: Overview

To see how to under-approximate computation, consider a control flow graph $G_k^{\texttt{main}}$. The under-approximation $\underline{G}_k^{\texttt{main}}$ 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 $\underline{G}_k^{\texttt{main}}$ hence is also a computation of $G_k^{\texttt{main}}$. Note that $\underline{G}_k^{\texttt{main}}$ is recursion-free. BasicAnalyzer is able to check the under-approximation $\underline{G}_k^{\texttt{main}}$.

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

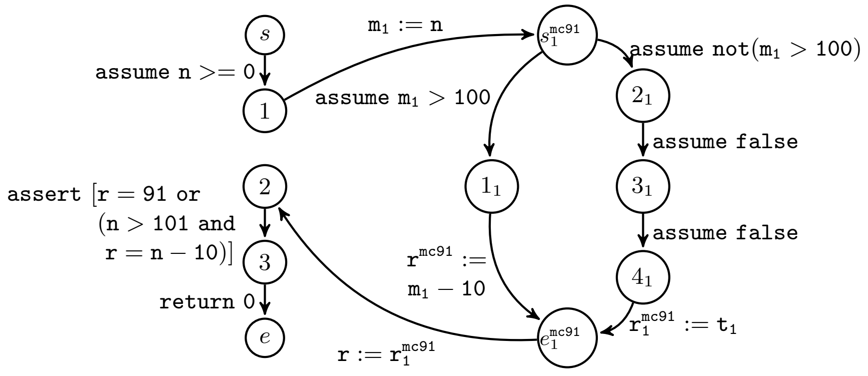


Fig. 2: Under-approximation of McCarthy 91

After function summaries are computed, Algorithm 1 verifies whether function summaries correctly specify computations of functions by invoking CheckSummary(P_k, S). The algorithm CheckSummary(P_k, S) checks this by constructing a recursion-free control flow graph $\tilde{G}^{\mathtt{f}}$ with additional assertions for each function `f` and verifying $\tilde{G}^{\mathtt{f}}$ with BasicAnalyzer. The control flow graph $\tilde{G}^{\mathtt{f}}$ is obtained by substituting function summaries for function calls. It is transformed from $G^{\mathtt{f}}$ 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 $\tilde{G}^{\mathtt{mc91}}$ with the function summary $S[\mathtt{mc91}] = \mathtt{not}(\mathtt{m} \geq 0)$. Observe that $\tilde{G}^{\mathtt{mc91}}$ is recursion-free. BasicAnalyzer is able to check $\tilde{G}^{\mathtt{mc91}}$ and invalidates this function summary.

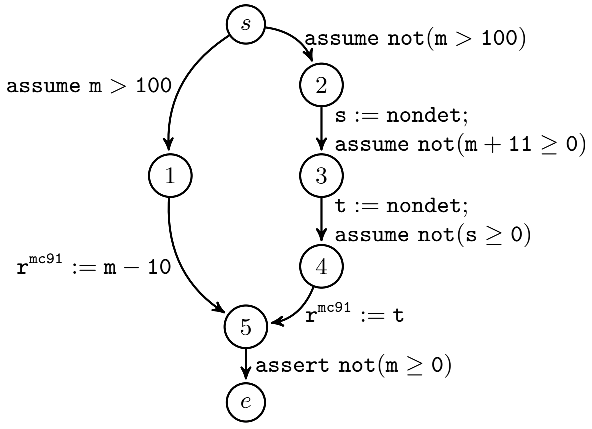


Fig. 3: Check Summary in McCarthy 91

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

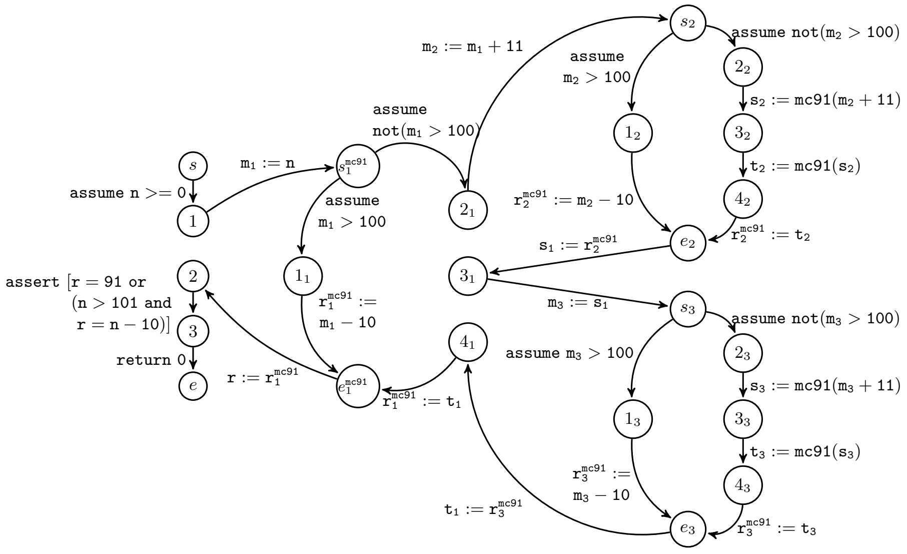

s
assume n >= 0
1
$\mathtt{m_1 := n}$
$s_1^{\mathtt{mc91}}$
assume not($\mathtt{m_1} > 100$)
assume $\mathtt{m_1} > 100$
2_1
$\mathtt{m_2 := m_1 + 11}$
s_2
assume $\mathtt{m_2} > 100$
assume not($\mathtt{m_2} > 100$)
2_2
$\mathtt{s_2 := mc91(m_2 + 11)}$
3_2
$\mathtt{t_2 := mc91(s_2)}$
4_2
1_2
$\mathtt{r_2^{mc91} := m_2 - 10}$
e_2
$\mathtt{r_2^{mc91} := t_2}$
$\mathtt{s_1 := r_2^{mc91}}$
3_1
$\mathtt{m_3 := s_1}$
s_3
assume not($\mathtt{m_3} > 100$)
assume $\mathtt{m_3} > 100$
2_3
$\mathtt{s_3 := mc91(m_3 + 11)}$
3_3
$\mathtt{t_3 := mc91(s_3)}$
4_3
1_3
$\mathtt{r_3^{mc91} := m_3 - 10}$
e_3
$\mathtt{r_3^{mc91} := t_3}$
$\mathtt{t_1 := r_3^{mc91}}$
4_1
$\mathtt{r_1^{mc91} := t_1}$
1_1
$\mathtt{r_1^{mc91} := m_1 - 10}$
$e_1^{\mathtt{mc91}}$
$\mathtt{r := r_1^{mc91}}$
2
assert [r = 91 or (n > 101 and r = n − 10)]
3
return 0
e

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

By Proposition 1, it follows that all assertions in $G^{\mathtt{main}}$ are satisfied. Moreover, by the semantics of the `assert` command, all assertions in the program are satisfied.

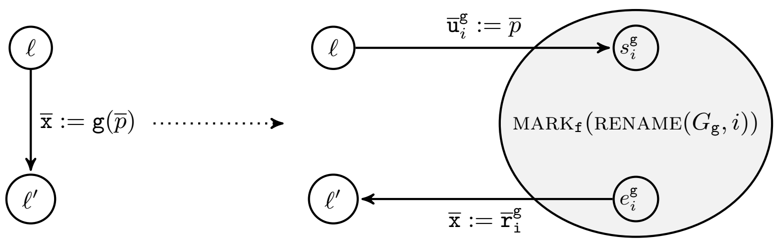

ℓ
$\overline{\mathtt{x}} := \mathtt{g}(\overline{p})$
ℓ'
ℓ
$\overline{\mathtt{u}}^{\mathtt{g}}_{i} := \overline{p}$
$s^{\mathtt{g}}_{i}$
MARK$_{\mathtt{f}}$(RENAME$(G_{\mathtt{g}}, i)$)
$e^{\mathtt{g}}_{i}$
ℓ'
$\overline{\mathtt{x}} := \overline{\mathtt{r}}^{\mathtt{g}}_{\mathtt{i}}$

Fig. 5: Unwinding Function Calls

4.1 Unwinding

We first define the rename function RENAME$(G^{\mathtt{f}}, i)$. It returns a CFG $\langle V_i, E_i, \mathrm{cmd}^{\mathtt{f}}_i, \overline{\mathtt{u}}^{\mathtt{f}}_i, \overline{\mathtt{r}}^{\mathtt{f}}_i, s_i, e_i\rangle$ obtained by first replacing every return command `return` $\overline{q}$ by assignments to return variables $\overline{\mathtt{r}}^{\mathtt{f}} := \overline{q}$ and then renaming all variables and locations in $G^{\mathtt{f}}$ with the index value i. The function UNWIND$(G^{\mathtt{f}})$ returns a CFG $K^{\mathtt{f}}$ obtained by replacing all function call edges in $G^{\mathtt{f}}$ with the CFG of the called function after renaming. In order to help extracting summaries from the $K^{\mathtt{f}}$, UNWIND$(G^{\mathtt{f}})$ annotates in $K^{\mathtt{f}}$ the outermost pair of the entry and exit locations s_i and e_i of each unwound function $\mathtt{g}$ with an additional superscript $\mathtt{g}$, i.e., $s^{\mathtt{g}}_i$ and $e^{\mathtt{g}}_i$ (Figure 5). The formal definition is given below.

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

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

Essentially, $G^{\mathtt{f}}$ and UNWIND$(G^{\mathtt{f}})$ 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 $G^{\mathtt{f}}$ before and after the function.

4.2 Under-approximation

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

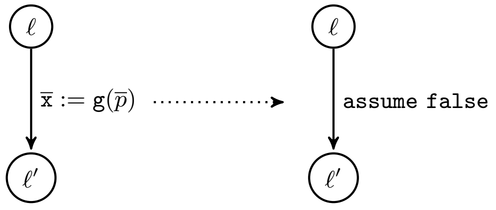


Fig. 6: Under-approximation

with `assume false` (Figure 6). That is,

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

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

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

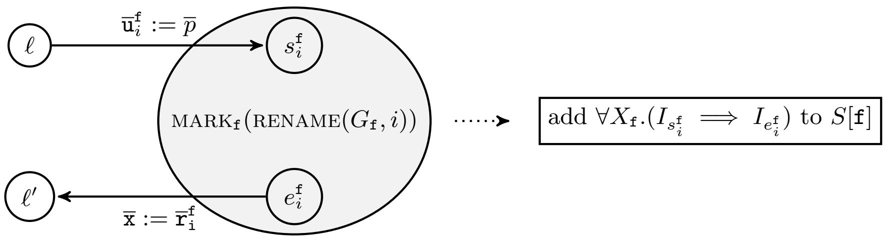
4.3 Computing Summaries

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

```
Input: $P_k$: a program; $\{I_\ell : \ell \in V\}$: an inductive invariant of $\underline{G}_k^{\mathtt{main}}$
Output: $S[\bullet]$: function summaries
foreach function f in the program $P_k$ do
    $S[\mathtt{f}] := \mathtt{true}$;
    foreach pair of locations $(s_i^{\mathtt{f}}, e_i^{\mathtt{f}}) \in V \times V$ do
        if $I_{s_i^{\mathtt{f}}}$ contains return variables of f then $S[\mathtt{f}] := S[\mathtt{f}] \wedge \forall X_{\mathtt{f}}.I_{e_i^{\mathtt{f}}}$ ;
    else $S[\mathtt{f}] := S[\mathtt{f}] \wedge \forall X_{\mathtt{f}}.(I_{s_i^{\mathtt{f}}} \implies I_{e_i^{\mathtt{f}}})$ ;
return $S[\bullet]$;
```

Algorithm 2: ComputeSummary($P_k, \Pi(\underline{G}_k^{\mathtt{main}}, \mathtt{true})$)

For each function `f` in the program P_k, we first initialize its summary $S[\mathtt{f}]$ to `true`. The set $X_{\mathtt{f}}$ contains all variables appearing in $\underline{G}_k^{\mathtt{main}}$ except the set of formal parameters and return variables of `f`. For each pair of locations $(s_i^{\mathtt{f}}, e_i^{\mathtt{f}}) \in V \times V$ in $\underline{G}_k^{\mathtt{main}}$, if the invariant of location $s_i^{\mathtt{f}}$ contains return variables of `f`, we update $S[\mathtt{f}]$ to the formula $S[\mathtt{f}] \wedge \forall X_{\mathtt{f}}.I_{e_i^{\mathtt{f}}}$. Otherwise, we update it to a less restricted version $S[\mathtt{f}] \wedge \forall X_{\mathtt{f}}.(I_{s_i^{\mathtt{f}}} \implies I_{e_i^{\mathtt{f}}})$ (Figure 7).



ℓ
$\overline{\texttt{u}}_i^{\texttt{f}} := \overline{p}$
$s_i^{\texttt{f}}$
$\text{MARK}_{\texttt{f}}(\text{RENAME}(G_{\texttt{f}}, i))$
add $\forall X_{\texttt{f}}.(I_{s_i^{\texttt{f}}} \implies I_{e_i^{\texttt{f}}})$ to $S[\texttt{f}]$
ℓ'
$e_i^{\texttt{f}}$
$\overline{\texttt{x}} := \overline{\texttt{r}}_{\texttt{i}}^{\texttt{f}}$

Fig. 7: Updating a Summary

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

The proposition holds because the only possible overlap of variables in Q and in $\overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})$ are the formal parameters $\overline{\texttt{u}}^{\texttt{f}}$. 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 $\overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})$.

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

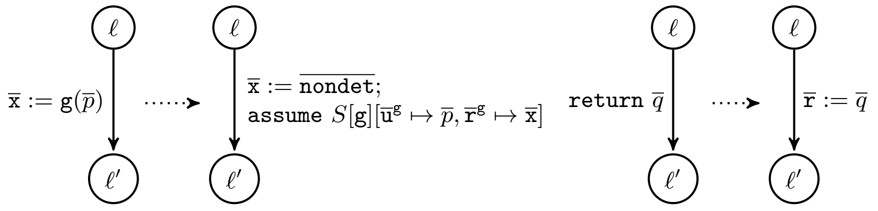
For each pair $(s_i^{\texttt{f}}, e_i^{\texttt{f}}) \in V \times V$, we consider two cases:

1. $I_{s_i^{\texttt{f}}}$ contains some return variables of $\texttt{f}$:
   In this case, the conjunct $\forall X_{\texttt{f}}.I_{e_i^{\texttt{f}}}$ is a part of $S[\texttt{f}]$, we then have

$$\cfrac{\cfrac{\cfrac{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|S[\texttt{f}]|)}{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|\forall X_{\texttt{f}}.I_{e_i^{\texttt{f}}}|)}\ \text{Postcondition Weakening}}{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{e_i^{\texttt{f}}}|)}\ \text{Postcondition Weakening}}{(|I_{s_i^{\texttt{f}}}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{e_i^{\texttt{f}}}|)}\ \text{Precondition Strengthening}$$

2. $I_{s_i^{\texttt{f}}}$ does not contain any return variables of $\texttt{f}$:
   In this case, the conjunct $\forall X_{\texttt{f}}.(I_{s_i^{\texttt{f}}} \implies I_{e_i^{\texttt{f}}})$ is a part of $S[\texttt{f}]$, we then have

$$\cfrac{\text{Prop. 4}\ \cfrac{}{(|I_{s_k^{\texttt{f}}}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{s_k^{\texttt{f}}}|)} \qquad \cfrac{\cfrac{\cfrac{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|S[\texttt{f}]|)}{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|\forall X_{\texttt{f}}.(I_{s_k^{\texttt{f}}} \implies I_{e_k^{\texttt{f}}})|)}}{(|\texttt{true}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{s_k^{\texttt{f}}} \implies I_{e_k^{\texttt{f}}}|)}}{(|I_{s_k^{\texttt{f}}}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{s_k^{\texttt{f}}} \implies I_{e_k^{\texttt{f}}}|)}}{(|I_{s_k^{\texttt{f}}}|)\ \overline{\texttt{r}}^{\texttt{f}} := \texttt{f}(\overline{\texttt{u}}^{\texttt{f}})\ (|I_{e_k^{\texttt{f}}}|)}$$



$\bar{\mathtt{x}} := \mathtt{g}(\bar{p})$
$\bar{\mathtt{x}} := \overline{\mathtt{nondet}};$
$\mathtt{assume}\ S[\mathtt{g}][\bar{\mathtt{u}}^{\mathtt{g}} \mapsto \bar{p}, \bar{\mathtt{r}}^{\mathtt{g}} \mapsto \bar{\mathtt{x}}]$
$\mathtt{return}\ \bar{q}$
$\bar{\mathtt{r}} := \bar{q}$
ℓ
ℓ'

Fig. 8: Instantiating a Summary

4.4 Checking Summaries

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

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

The control flow graph $\hat{G}_{k,S}^{\mathtt{f}}$ replaces every function call in $G_k^{\mathtt{f}}$ by instantiating a function summary (Figure 8). Using the Hoare Logic proof rule for recursive functions [22], we have the following proposition:

Proposition 6. *Let $G_k^{\mathtt{f}} = \langle V, E, cmd^{\mathtt{f}}, \bar{\mathtt{u}}^{\mathtt{f}}, \bar{\mathtt{r}}^{\mathtt{f}}, s, e\rangle$ 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 $(\!|\mathtt{true}|\!)\ \hat{G}_{k,S}^{\mathtt{g}}\ (\!|S[\mathtt{g}]|\!)$ for every function* g *in P, then $(\!|\mathtt{true}|\!)\ \bar{\mathtt{r}}^{\mathtt{f}} := \mathtt{f}(\bar{\mathtt{u}}^{\mathtt{f}})\ (\!|S[\mathtt{f}]|\!)$.*

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

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

Corollary 1. *Let $G_k^{\mathtt{f}} = \langle V, E, cmd^{\mathtt{f}}, \bar{\mathtt{u}}^{\mathtt{f}}, \bar{\mathtt{r}}^{\mathtt{f}}, s, e\rangle$ 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* BasicChecker*$(\tilde{G}_{k,S}^{\mathtt{g}})$ returns Pass for every function* g *in P, then $(\!|\mathtt{true}|\!)\ \bar{\mathtt{r}}^{\mathtt{f}} := \mathtt{f}(\bar{\mathtt{u}}^{\mathtt{f}})\ (\!|S[\mathtt{f}]|\!)$.*

```
Input: $P_k$ : an unwound program; $S[\bullet]$ : an array of function summaries
Output: true if all function summaries are valid; false otherwise
foreach function $G_k^{\mathtt{g}} \in P_k$ do
    if BasicChecker($\tilde{G}_{k,S}^{\mathtt{g}}$) $\neq$ 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 $\mathit{Pass}(\mathit{\Pi}(\underline{G}_k^{\mathtt{main}}, \mathtt{true}))$ and $S[\bullet]$ on the input control flow graph $G^{\mathtt{main}} = \langle V, E, \mathrm{cmd}^{\mathtt{main}}, \overline{\mathtt{u}}^{\mathtt{main}}, \overline{\mathtt{r}}^{\mathtt{main}}, s, e\rangle$. Let $\underline{G}_k^{\mathtt{main}} = \langle \underline{V}_k, \underline{E}_k, \underline{\mathrm{cmd}}_k^{\mathtt{main}}, \overline{\mathtt{u}}^{\mathtt{main}}, \overline{\mathtt{r}}^{\mathtt{main}}, s, e\rangle$ and $\mathit{\Pi}(\underline{G}_k^{\mathtt{main}}, \mathtt{true}) = \{\underline{I}_\ell : \ell \in \underline{V}_k\}$. By the definition of inductive invariants, we have $(\!|\underline{I}_\ell|\!)\ \underline{\mathrm{cmd}}_k^{\mathtt{main}}(\ell, \ell')\ (\!|\underline{I}_{\ell'}|\!)$ for every $(\ell, \ell') \in \underline{E}_k$. Moreover, $V \subseteq \underline{V}_k$ since $\underline{G}_k^{\mathtt{main}}$ is obtained by unwinding $G^{\mathtt{main}}$. Define $\mathit{\Gamma}(G^{\mathtt{main}}, \mathtt{true}) = \{\underline{I}_\ell \in \mathit{\Pi}(\underline{G}_k^{\mathtt{main}}, \mathtt{true}) : \ell \in V\}$. We claim $\mathit{\Gamma}(G^{\mathtt{main}}, \mathtt{true})$ is in fact an inductive invariant for $G^{\mathtt{main}}$.

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

$$(\!|\underline{I}_\ell|\!)\ \overline{\mathtt{x}} := \mathtt{f}(\overline{p})\ (\!|\underline{I}_{\ell'}|\!) \text{ or, equivalently, } (\!|\underline{I}_\ell|\!)\ \overline{\mathtt{u}}^{\mathtt{f}} := \overline{p};\ \overline{\mathtt{r}}^{\mathtt{f}} := \mathtt{f}(\overline{\mathtt{u}}^{\mathtt{f}});\ \overline{\mathtt{x}} := \overline{\mathtt{r}}^{\mathtt{f}}\ (\!|\underline{I}_{\ell'}|\!)$$

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

$$\dfrac{\dfrac{(\!|\underline{I}_\ell|\!)\ \overline{\mathtt{u}}^{\mathtt{f}} := \overline{p}\ (\!|\underline{I}_{s_k^{\mathtt{f}}}|\!) \qquad (\!|\underline{I}_{s_k^{\mathtt{f}}}|\!)\ \overline{\mathtt{r}}^{\mathtt{f}} := \mathtt{f}(\overline{\mathtt{u}}^{\mathtt{f}})\ (\!|\underline{I}_{e_k^{\mathtt{f}}}|\!) \qquad (\!|\underline{I}_{e_k^{\mathtt{f}}}|\!)\ \overline{\mathtt{x}} := \overline{\mathtt{r}}^{\mathtt{f}}\ (\!|\underline{I}_{\ell'}|\!)}{(\!|\underline{I}_\ell|\!)\ \overline{\mathtt{u}}^{\mathtt{f}} := \overline{p};\ \overline{\mathtt{r}}^{\mathtt{f}} := \mathtt{f}(\overline{\mathtt{u}}^{\mathtt{f}});\ \overline{\mathtt{x}} := \overline{\mathtt{r}}^{\mathtt{f}}\ (\!|\underline{I}_{\ell'}|\!)}}{(\!|\underline{I}_\ell|\!)\ \overline{\mathtt{x}} := \mathtt{f}(\overline{p})\ (\!|\underline{I}_{\ell'}|\!)}$$

5 Experiments

A prototype tool of our approach has been implemented with CPAChecker 1.2.11-svcomp14b[3] as the underlying intraprocedural analyzer. In addition, because CPAChecker does not support universal quantifiers in the expression of an `assume` command, we used Redlog [19] for quantifier elimination. To evaluate our tool, we performed experiments with all the benchmarks from the **recursive** category in the 2014 Competition on Software Verification (SV-COMP

[3] 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`.

2014) [9] 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 solid evidence that our approach not only extends program analyzers to handle recursion but also provides comparable effectiveness.

Our tool was compared with four participants of SV-COMP 2014, namely Blast 2.7.2[4] [5], CBMC 4.5-sv-comp-2014 [8] with a wrapper cbmc-wrapper.sh[5], Ultimate Automizer [13], and Ultimate Kojak [21]. 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 comprise 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 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[6], 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.

6 Related Works

In [14, 15], a program transformation technique for checking context-bounded concurrent programs to sequential analysis is developed. Numerous intrapro-

[4] We use the arguments **-alias empty -enable-recursion -noprofile -cref -sv-comp -lattice -include-lattice symb -nosserr** with Blast.

[5] The wrapper cbmc-wrapper.sh is provided by CBMC 4.5-sv-comp-2014, which is a special version for SV-COMP 2014.

[6] 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. (Time in sec.)

Program	Our Tool		Ultimate Automizer	Ultimate Kojak	CBMC 4.5	Blast 2.7.2
	k	Time	Time	Time	Time	Time
Ackermann01_true.c	1	6.5	T.O.	T.O.	850.0	E
Ackermann02_false.c	4	57.3	4.2	T.O.	1.0	E
Ackermann03_true.c		T.O.	T.O.	T.O.	850.0	E
Ackermann04_true.c		T.O.	T.O.	T.O.	850.0	E
Addition01_true.c	2	14.1	T.O.	T.O.	850.0	E
Addition02_false.c	2	9.9	3.7	3.5	0.3	4.0
Addition03_false.c		T.O.	T.O.	T.O.	850.0$^{!}$	E
EvenOdd01_true.c	1	2.9$^{!}$	T.O.	T.O.	1.3	0.1$^{!}$
EvenOdd03_false.c	1	2.9	3.2	3.2	0.1	0.1
Fibonacci01_true.c	6	348.4	T.O.	T.O.	850.0	E
Fibonacci02_true.c		T.O.	60.7	72.1$^{?}$	0.8	E
Fibonacci03_true.c		T.O.	T.O.	T.O.	850.0	E
Fibonacci04_false.c	5	107.3	7.4	8.2	0.4	E
Fibonacci05_false.c		T.O.	128.9	23.2	557.2	E
gcd01_true.c	1	6.6	5.4	7.3	850.0	16.1$^{!}$
gcd02_true.c		T.O.	T.O.	T.O.	850.0	E
McCarthy91_false.c	1	2.8	3.2	3.1	0.3	0.1
McCarthy91_true.c	2	12.5	81.3	6.8	850.0	16.2$^{!}$
MultCommutative_true.c		T.O.	T.O.	T.O.	850.0	E
Primes_true.c		T.O.	T.O.	T.O.	850.0	E
recHanoi01_true.c		T.O.	T.O.	T.O.	850.0	E
recHanoi02_true.c	1	5.6	T.O.	T.O.	0.7	1.9$^{!}$
recHanoi03_true.c		T.O.	T.O.	T.O.	0.7	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

cedural analysis techniques have been developed over the years. Many tools are in fact freely available (see, for instance, Blast [5], CPAChecker [6], and UFO [2]). Interprocedural analysis techniques are also available (see [20, 4, 10, 12, 11, 18] for a partial list). Recently, recursive analysis attracts new attention. The Competition on Software Verification adds a new category for recursive programs in 2014 [9]. Among the participants, CBMC [8], Ultimate Automizer [13], and Ultimiate Kojak [21] are the top three tools for the **recursive** category.

Inspired by Whale [1], we use inductive invariants obtained from verifying under-approximation as candidates of summaries. Also, similar to Whale, we apply a Hoare logic proof rule for recursive calls from [22]. 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.

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. In particular, 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.

Acknowledgment

This work was partially supported by Ministry of Science and Technology under grant numbers 100-2221-E-002 -122 -, 102-2221-E-001 -017 -, 102-2221-E-001 -018 -, and the postdoctoral fellow program of Academia Sinica, Taiwan.

References

1. Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.

2. 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.
3. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
4. Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264, 2001.
5. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.
6. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
7. Edmund M. Clarke, Himanshu Jain, and Nishant Sinha. Grand challenge: Model check software. In *VISSAS*, pages 55–68, 2005.
8. Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.
9. Competition on software verification. `http://sv-comp.sosy-lab.org/2014`.
10. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *ESOP*, pages 21–30, 2005.
11. Coverity. `http://www.coverity.com/`.
12. 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.
13. 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.
14. Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, pages 37–51, 2008.
15. 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.
16. Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *J. ACM*, 17(3):555–569, 1970.
17. Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999. ISBN 978-3-540-65410-0.
18. Polyspace. `http://www.mathworks.com/products/polyspace/`.
19. Redlog. `http://www.redlog.eu/`.
20. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
21. Ultimate Kojak. `http://ultimate.informatik.uni-freiburg.de/kojak/`.
22. David von Oheimb. Hoare logic for mutual recursion and local variables. In *FSTTCS*, pages 168–180, 1999.