

國立臺灣大學電機資訊學院電子工程學研究所
碩士論文

Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

以PAC安全性保證實作程式測試
Program Testing with PAC Guarantees

李宗儒
Lii, Tsung-Ju

指導教授：王凡博士
Advisor: Farn Wang, Ph.D.

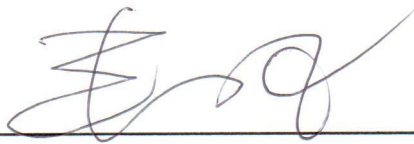
中華民國 105 年 5 月
May 2016

國立臺灣大學碩士學位論文
口試委員會審定書

以程式變換方法協助程式分析器驗證遞迴程式
Verifying Recursive Program via
Source-to-Source Program Transformation

本論文係謝橋君（R02943142）在國立臺灣大學電子工程學研究所完成之碩士學位論文，於民國 104 年 7 月 3 日承下列考試委員審查通過及口試及格，特此證明

口試委員：



（指導教授）

王柏堯

戴頻耀

陳育亨

江介宏

系主任、所長



Verifying Recursive Program via Source-to-Source Program Transformation

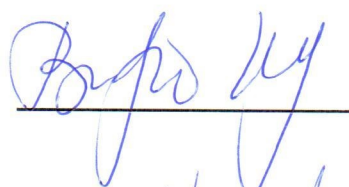

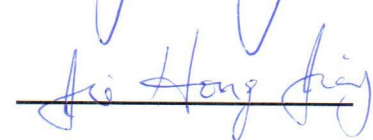
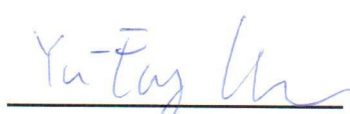
By
Hsieh, Chiao

THESIS

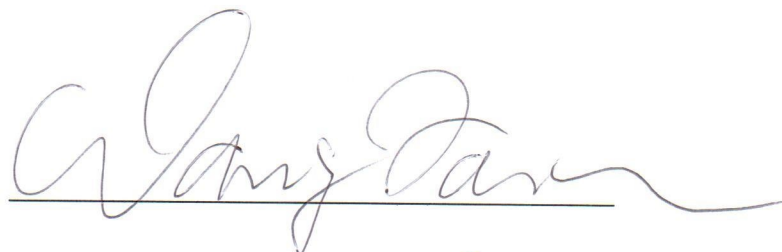
Submitted in partial fulfillment of the requirement
for the degree of Master of Science in Electronics Engineering
at National Taiwan University
Taipei, Taiwan, R.O.C.

July 2015

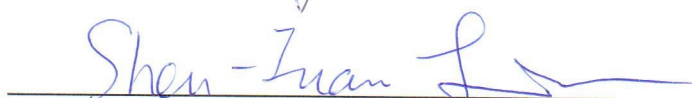
Approved by :

Advised by :



Approved by Director :



Acknowledgements

I should like to thank Prof. Yu-Fang Chen and Prof. Bow-Yaw Wang at Formal Methods Laboratory in Academia Sinica. Your guidance and instructions helped me gradually develop the whole framework since I was a full-time research assistant in FM Lab. Your solid training and teaching also forced me to establish my knowledge of Software Verification and Formal Methods. In particular, the opportunity and the experience you gave me to attend international conferences surely is very enlightening though quite frightening. In addition, I am specially grateful to and surprised by the assistance from Dr. Ming-Hsien Tsai on the implementation. Your efficiency and proficiency on software programming skills are simply remarkable and incomparable. Without your assistance, I am sure I cannot finish the implementation on my own, let alone participating the competition.

I also learned a lot and pretty enjoyed my life in Software Testing Laboratory in National Taiwan University. Prof. Farn Wang is a great role model as a academic researcher. His ambition on building an industrial strength software testing tool is respectable. The laboratory members and assistants are just adorable. You guys inspired me in a unique way. Even when we were both struggling for homeworks, reports, exams, and finally the graduation, you guys still found many ways to relax and relief the tension among the whole lab. "Work Hard, Play Hard" is the spirit I learned from you. You guys rock.

Finally, my appreciation to my parents and siblings is beyond words. Thanks for your constant nagging on me about my physically and mentally unhealthy lifestyle as well as trying to stop my 26-year-without-girlfriend achievement. Your concerns and considerations kept reminding me that, even if I might fail so hard and be so frustrated, I still can go home, where I can relax and recover until the next challenge comes.

Thank you all.

National Taiwan University, Taipei
July, 2015

謝橋 Hsieh, Chiao

摘要

在程序分析 (Program Analysis) 領域中，分析遞迴程式 (Recursive Program) 是一項難以處理的課題。現有程式分析工具往往因為無法處理程式中的遞迴函式，既而忽略部分程式碼不進行分析，或甚至拒絕檢驗遞迴程式。本篇論文針對遞迴程式的分析提出新的觀點與分析方法；作者認為與其重新設計演算法且實作新的分析器，不如改進現有的程式分析工具。但更改他人實作的軟體工具並非易事，因此本文提出的方法將現有的工具視為黑箱不做任何更動，反而是利用程式變換方法 (Program Transformation) 將待分析的遞迴程式轉換成無遞迴的程式，再交給黑箱分析工具進行檢驗；並使用黑箱工具提供被分析程式中的不變量 (Invariants)，進而證明原遞迴程式的正確性。

本篇作者與實驗室團隊受惠於程式分析工具 CPACHECKER，實作出此演算法的雛型 CPAREC，且參與 2015 年度軟體驗證競賽 (Competition on Software Verification)；和其它頂尖實驗室所開發工具相較，我們的工具針對遞迴程式進行分析時，亦表現出相當的效率及效能，獲得第三名的佳績。

關鍵字：軟體驗證、程序分析、靜態分析、遞迴程式、程式變換方法

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 non-deterministic 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 named CPAREC using the recursion-free program analyzer CPACHECKER. Under the comparison with other program analyzers on the benchmarks in the 2014 and 2015 Competitions on Software Verification, our tool shows competitive efficiency and effectiveness on verifying programs with recursion.

Key words: CPAREC, Software Verification, Program Analysis, Static Analysis, Recursion, Program Transformation

Contents

口試委員會審定書	i
Oral Examination Approval Form	ii
Acknowledgements	iii
摘要	iv
Abstract	v
1 Introduction	1
2 Related Works	4
3 Preliminaries	5
3.1 Program Model	5
3.2 Hoare Logic and Program Analysis	7
4 Overview	9
5 Proving via Transformation	13
5.1 Unwinding	13
5.1.1 Function Inlining Method	14
5.1.2 Function Duplicating Method	15
5.2 Under-approximation	18
5.2.1 Approximating Functions	18

5.2.2	Approximating Programs from Function Inlining Method	19
5.2.3	Approximating Programs from Function Duplicating Method	19
5.3	Computing Summaries	20
5.4	Checking Summaries	22
5.5	Correctness	24
6	Experiments	26
6.1	Implementation	26
6.2	Experiments	27
7	Conclusion	32

List of Figures

3.1	McCarthy 91	7
4.1	Under-approximation of McCarthy 91	11
4.2	Check Summary in McCarthy 91	11
4.3	Unwinding McCarthy 91	12
5.1	Unwinding Function Calls	14
5.2	Under-approximation	19
5.3	Updating a Summary	21
5.4	Instantiating a Summary	23
6.1	The Execution Flow of CPA _{REC}	27
6.2	Accumulated Score over Time	31

List of Tables

6.1	Scoring scheme in SV-COMP 2014.	28
6.2	Experimental results of verifying programs in the recursive category of the 2014 Competition on Software Verification. (Time in sec.)	29
6.3	Scoring scheme in SV-COMP 2015.	30
6.4	Partial competition results of the recursive category of the 2015 Compe- tition on Software Verification.	31

Chapter 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 [1]. 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 intraprocedural (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 recursion-free ones and invokes an intraprocedural program analyzer to verify properties about the generated recursion-free 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 recursion-free program that *under-approximates* the behaviors of the original

and sends the under-approximation to an intraprocedural 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 recursion-free programs with purported function summaries. It finally checks if purported function summaries are correct by sending these recursion-free programs to the analyzer.

Compared with other analysis algorithms for recursive programs, our approach is very lightweight. It only performs syntactic transformation and requires standard functionalities from underlying intraprocedural 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 [2]. 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 [3]. Encouraged by the result, we submitted and successfully published our approach on Static Analysis Symposium [4]. We further participated the 2015 Competition on Software Verification [5] under the name CPAREC [6].

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

improvements are discussed in Chapter 7.

Chapter 2

Related Works

Numerous intraprocedural analysis techniques have been developed over the years. Many tools are in fact freely available (see, for instance, `BLAST` [7], `CPACHECKER` [2], and `UFO` [8]). Interprocedural analysis techniques are also available (see [9]–[14] for a partial list). Recently, recursive analysis attracts new attention. The Competition on Software Verification adds a new category for recursive programs in 2014 [3]. Among the participants, `CBMC` [15], `ULTIMATE AUTOMIZER` [16], and `ULTIMATE KOJAK` [17] are the top three tools for the **recursive** category.

Inspired by `WHALE` [18], 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 [19]. 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.

Aside from recursion analysis, using program transformation to reduce program features is not a new concept. We learned this concept particularly from [20], [21], where a program transformation technique for checking context-bounded concurrent programs to sequential analysis is developed.

Chapter 3

Preliminaries

3.1 Program Model

We consider a variant of the WHILE language from [22, Sec. 1.2]:

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

Vars denotes the set of *program variables*, and $\text{Vars}' = \{\mathbf{x}' : \mathbf{x} \in \text{Vars}\}$ where \mathbf{x}' represents the new value of \mathbf{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 program P is simply a set of functions. Each function f in the program P is represented as a *control flow graph* (CFG) $G^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$ 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 $\text{cmd}^f(\ell, \ell')$, \bar{u}^f and \bar{r}^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^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 [23].

Figure 3.1 shows control flow graphs for the McCarthy 91 program from [24]. 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 3.1a). The `mc91` function branches on whether the variable `m` is greater than 100. If so, it returns `m - 10`. Otherwise, `mc91(m)` stores the result of `mc91(m + 11)` in `s`, and returns the result of `mc91(s)` (Figure 3.1b). Observe that a conditional branch is modeled with the `assume` command in the figure. Loops can be modeled similarly.

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

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

where I' is obtained by replacing every $x \in \text{Vars}$ in I with $x' \in \text{Vars}'$, and $\tau_{\text{cmd}^f(\ell, \ell')}$

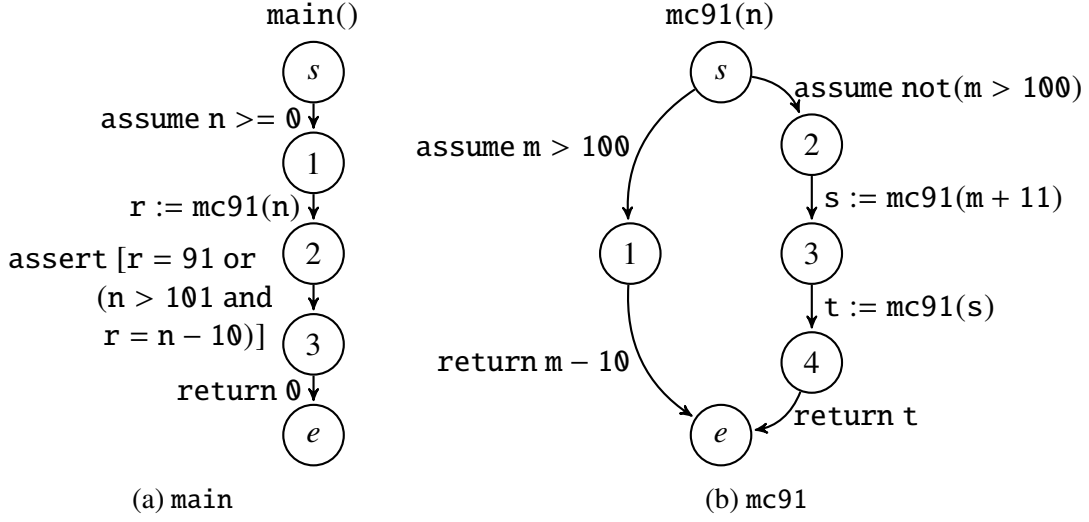


Figure 3.1: McCarthy 91

specifies the semantics of the command $\text{cmd}^f(\ell, \ell')$. 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 f starts from a state satisfying I_0 . For every $\ell \in V$, G^f must arrive in a state satisfying I_ℓ when the computation reaches ℓ .

3.2 Hoare Logic and Program Analysis

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. For intraprocedural commands, we extend the standard proof rules for partial correctness in [25, Sec. 9.2] 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 aborts the computation if the given expression is not implied by the precondition. No postcondition can be guaranteed in such a case. For interprocedural analysis, we adopt the proof rules from [19] for (recursive) function calls. In addition,

there are other rules, e.g., the rules for unconditional jumps [26]. However, those proof rules are not involved in our proof for correctness, we consider them to be out of scope.

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 3.1 *Let $G^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$ 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 which 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 [2], BLAST [7], UFO [8], ASTRÉE [11], etc. Our goal is to check recursive programs by using a recursion-free inductive program analyzer as a black box.

Chapter 4

Overview

Let `BASICANALYZER` denote a recursion-free inductive program analyzer, and let a program $P = \{G^{\text{main}}\} \cup \{G^f : f \text{ is a function}\}$ consist of the CFGs of the `main` function and functions that may be invoked (transitively) from `main`. Since functions without recursive calls 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^{main} into a recursion-free program $\underline{G}^{\text{main}}$. The program $\underline{G}^{\text{main}}$ under-approximates the computation of G^{main} . That is, every computation of $\underline{G}^{\text{main}}$ is also a computation of G^{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 recursion-free under-approximation $\underline{G}^{\text{main}}$. Our algorithm computes 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).

To see how to under-approximate computation, consider a control flow graph G_k^{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 4.1). The substitution effectively blocks all recursive invocations. Any computation of $\underline{G}_k^{\text{main}}$ hence is also a computation

Input: A program $P = \{G^{\text{main}}\} \cup \{G^f : f \text{ is a function}\}$

$k := 0;$

$P_0 := P;$

repeat

$k := k + 1;$

$P_k := \text{unwind every CFG in } P_{k-1};$

switch BASICANALYZER($\underline{G}_k^{\text{main}}$) **do**

case $\text{Pass}(\Pi(\underline{G}_k^{\text{main}}, \text{true}))$: **do**

$S := \text{ComputeSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$

case Error : **do return** Error ;

$\text{complete?} := \text{CheckSummary}(P_k, S);$

until complete? ;

return $\text{Pass}(\Pi(\underline{G}_k^{\text{main}}, \text{true})), S;$

Algorithm 1: Overview

of G_k^{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 computes 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 $\text{ComputeSummary}(P_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$ computes summaries S by inspecting the inductive invariant $\Pi(\underline{G}_k^{\text{main}}, \text{true})$ (Section 5.3).

After function summaries are computed, 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 recursion-free control flow graph \tilde{G}^f with additional assertions for each function f and verifying \tilde{G}^f with BASICANALYZER. The control flow graph \tilde{G}^f is obtained by substituting function summaries for function calls. It is transformed from G^f by the following three steps:

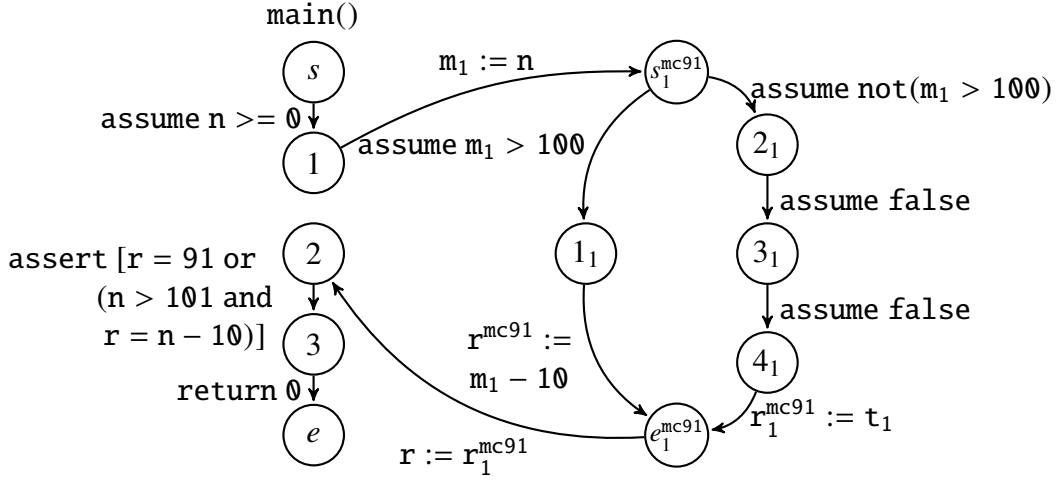


Figure 4.1: Under-approximation of McCarthy 91

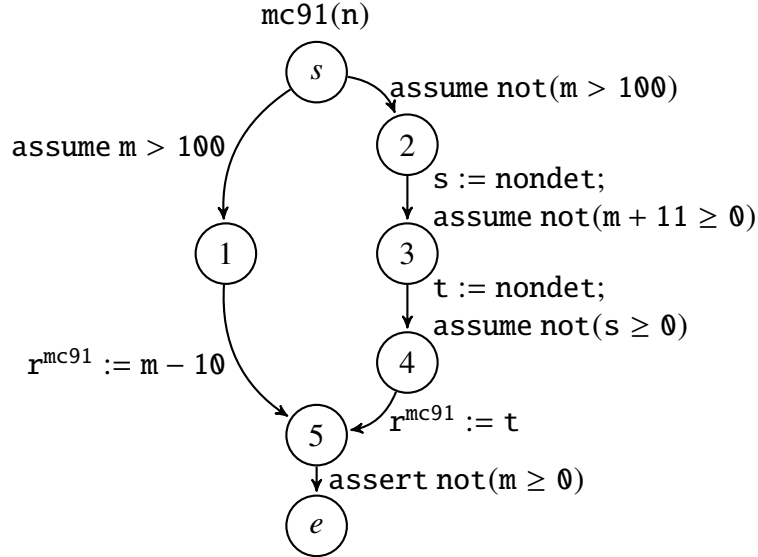


Figure 4.2: Check Summary in McCarthy 91

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

In order to refine function summaries, our algorithm unwinds recursive functions as usual. More precisely, consider a recursive function f with formal parameters \bar{u}^f and

Chapter 5

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 5.1 *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 3.1, it follows that all assertions in G^{main} are satisfied. Moreover, by the semantics of the `assert` command, all assertions in the program are satisfied.

5.1 Unwinding

As introduced in Chapter 4, our algorithm produces more and more accurate under-approximations through unwinding the program. An important feature of unwinding is that it should keep the behavior of unwound program being the same as the original one. In this section, we introduce two function unwinding methods accompanied with simple proofs of the equivalence between the original and unwound functions. We also devise different strategies to produce unwound programs when using different function unwinding methods.

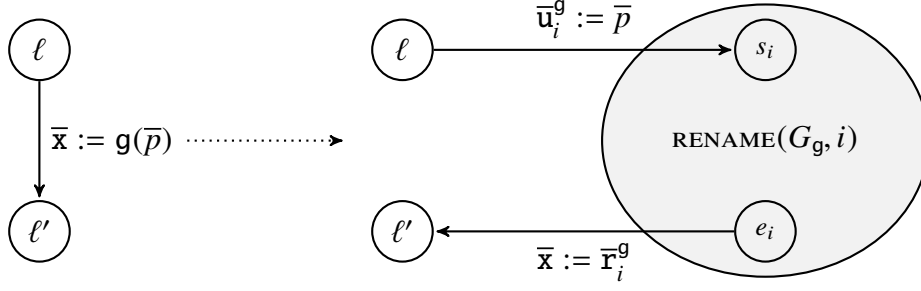


Figure 5.1: Unwinding Function Calls

5.1.1 Function Inlining Method

The function inlining method directly follows the concept of inline function in C/C++ language and compiler optimizations. It expands the body at the location of a function call. We can achieve this in our program model. The function $\text{UNWIND}(G^f)$ returns a CFG K^f obtained by replacing all function call edges in G^f with the CFGs of the called functions (Figure 5.1). The formal definition is given below.

We first define the function $\text{RENAME}(G^f, i)$. It returns a CFG obtained by first replacing every return command $\text{return } \bar{q}$ with assignments to return variables $\bar{r}^f := \bar{q}$ and then renaming all variables and locations in G^f with the given index value i . The function $\text{ADDINDEX}(c, i)$ is used to rename all variables in a command c with index i . Formally,

$$\begin{aligned} \text{RENAME}(\langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle, i) &= \langle V_i, E_i, \text{cmd}_i^f, \bar{u}_i^f, \bar{r}_i^f, s_i, e_i \rangle \text{ such that} \\ V_i &= \{\ell_i : \ell \in V\} \\ E_i &= \{(\ell_i, \ell'_i) : (\ell, \ell') \in E\} \\ \text{cmd}_i^f(\ell_i, \ell'_i) &= \begin{cases} \text{ADDINDEX}(\bar{r}^f := \bar{q}, i) & \text{if } \text{cmd}^f(\ell, \ell') = \text{return } \bar{q} \\ \text{ADDINDEX}(\text{cmd}^f(\ell, \ell'), i) & \text{otherwise} \end{cases} \end{aligned}$$

Then, given a CFG $G^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$, we use $\hat{E} = \{(\ell, \ell') \in E : \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p}))\}$ 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. Finally, we provide the definition of $\text{UNWIND}(G^f)$.

$$\text{UNWIND}(G^f) = \langle V_u, E_u, \text{cmd}_u^f, \bar{u}^f, \bar{r}^f, s, e \rangle \text{ where}$$

$$\begin{aligned}
V_u &= V \cup \bigcup \{V_i : (\ell, \ell') \in \hat{E} \wedge \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \wedge \text{IDX}(\ell, \ell') = i \\
&\quad \wedge \text{RENAME}(G^g, i) = \langle V_i, E_i, \text{cmd}_i^g, \bar{u}_i^g, \bar{r}_i^g, s_i, e_i \rangle\} \\
E_u &= E \setminus \hat{E} \cup \bigcup \{E_i \cup \{(\ell, s_i), (e_i, \ell')\} : (\ell, \ell') \in \hat{E} \wedge \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \\
&\quad \wedge \text{IDX}(\ell, \ell') = i \wedge \text{RENAME}(G^g, i) = \langle V_i, E_i, \text{cmd}_i^g, \bar{u}_i^g, \bar{r}_i^g, s_i, e_i \rangle\} \\
\text{cmd}_u^f(\ell, \ell') &= \begin{cases} (\bar{u}_i^g := \bar{p}) & \text{if } (\ell, \ell') = (\ell, s_i) \\
(\bar{x} := \bar{r}_i^g) & \text{if } (\ell, \ell') = (e_i, \ell') \\
\text{cmd}_i^g(\ell, \ell') & \text{if } (\ell, \ell') \in E_i \\
\text{cmd}^f(\ell, \ell') & \text{otherwise} \end{cases}
\end{aligned}$$

Proposition 5.2 *Let G^f be a control flow graph. P and Q are logic formulas 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.

Unwinding Program

Considering the UNWIND , it transforms a single CFG into another single CFG with the same behavior but more program variables. Following our program, the program starts at `main` function. Therefore, to unwind the program in each iteration of Algorithm 1, we can simply apply UNWIND on G^{main} function repeatedly. In other words,

$$\begin{aligned}
&\text{Initially, } P_0 = P \therefore G_0^{\text{main}} = G^{\text{main}} \\
&G_k^{\text{main}} = \text{UNWIND}(G_{k-1}^{\text{main}}) \\
&P_k = P_{k-1} \setminus \{G_{k-1}^{\text{main}}\} \cup \{G_k^{\text{main}}\}
\end{aligned}$$

5.1.2 Function Duplicating Method

The previously mentioned method comes with an obvious drawback that it will cause exponential increase in size of the unwound program. Considering the McCarthy 91 function, the number of recursive calls in the program doubles in the unwound function at each iteration of Algorithm 1, and the program size doubles at next iteration since every

function call is inlined, i.e., transformed to a copy of the function body. Therefore, we devised another method $\text{UNWIND}_P(G^f, k)$ that duplicates functions in program P instead of inlining function calls to prevent exponential growth. Note that the $\text{UNWIND}_P(G^f, k)$ also preserves the behavior of original function f in the modified program P after unwinding.

The procedure of the new method is simple. We give a function body G^f which contains multiple function call edges to the same functions g as an example. In the old method, each function call to g is replaced by almost the same function body G^g except being indexed during unwinding, and therefore the program size grows exponentially. The new method creates a new function g_k with a given index value k . The function g_k has exactly the same function body G^g . Those calls to g are then replaced with calls to g_k . Only one additional function body is introduced for multiple function calls. Notice that, since there are a modified version of G^f and many duplicated functions G^{g_k} , UNWIND_P returns a pair of a function and a set of functions instead of one function. The formal definition is given as follows.

First, we define $\text{DUPLICATE}_P(g, k)$ to copy g in the program P . Given k indicating the k -th iteration in Algorithm 1, we create a new function g_k with the body G^{g_k} which is almost the same as G^g except being indexed via RENAME .

$$\text{DUPLICATE}_P(G^g, k) = G^{g_k} \text{ where } g_k \notin P \wedge G^{g_k} = \text{RENAME}(G^g, k)$$

Then, to unwind a given CFG G^f , we produce copies of the functions called by f and replace all call edge by calls to the duplication. Formally, the definition of $\text{UNWIND}_P(G^f)$ is as below. (Please refer to § 5.1.1 for definitions of the symbols).

$$\begin{aligned} \text{UNWIND}_P(G^f, k) &= \langle G_k^f, \text{DUP}_k^f \rangle \text{ where} \\ \text{DUP}_k^f &= \{G^{g_k} : \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \wedge G^{g_k} = \text{DUPLICATE}_P(G^g, k)\} \\ G_k^f &= \langle V, E, \text{cmd}_u^f, \bar{u}^f, \bar{r}^f, s, e \rangle \\ \text{cmd}_u^f(\ell, \ell') &= \begin{cases} (\bar{x} := g_k(\bar{p})) & \text{if } \text{cmd}^f(\ell, \ell') = (\bar{x} := g(\bar{p})) \\ & \wedge G^{g_k} = \text{DUPLICATE}_P(G^g, k) \\ \text{cmd}^f(\ell, \ell') & \text{otherwise} \end{cases} \end{aligned}$$

Notice that, if function f contains multiple calls to the same function g , the CFG G^{g_k} is not introduced multiple times by set union. This is the primary reason that leads to less increase in program size.

Proposition 5.3 *Let f be a function without self or mutual recursion and G^f be its control flow graph. P and Q are logic formulas with free variables over program variables of G^f . $\langle P \rangle G^f \langle Q \rangle$ if and only if $\langle P \rangle \text{UNWIND}_P(G^f, k) \langle Q \rangle$.*

In this method, the only difference between G^f and $\text{UNWIND}_P(G^f, k)$ is that each function call to g is substituted with call to g_k . Hence, the proposition should trivially hold if the behavior of g and g_k have the same behavior. Under the premise that f is a function without recursion, g_k won't introduce new recursion since it does not belong to original program P . Therefore, g_k and g simply behave the same because $G^{g_k} = \text{RENAME}(G^g, k)$. Without the premise, g can directly or indirectly call f , and introducing g_k complicates the mutual recursion further. Thus, proving the proposition without the premise is much harder. Since the stronger proposition is not necessary for our work, we don't provide further proof here.

Unwinding Program

Unwinding the program with UNWIND_P is very different from the one with UNWIND because we have to add duplication of functions into the program P . Repeatedly applying UNWIND_P on main does not help this time since it only copies the functions called by main again and again. In our approach, we apply unwinding on main in the first iteration, and on only those duplications of functions in following iterations. This is crucial because the main function and the duplications are guaranteed to be not involved in recursive calls and fulfill the premise in Proposition 5.3.

Since there are multiple functions being unwound in an iteration, we extend UNWIND_P to handle a set of functions by merging all results into a set. The extended method therefore

returns a pair of sets of functions. Formally, given a set of CFGs S ,

$$\text{UNWIND}_P(S, k) = \langle GS_k, DUPS_k \rangle \text{ where}$$

$$GS_k = \{G_k^f : G^f \in S \wedge \text{UNWIND}_P(G^f, k) = \langle G_k^f, DUP_k^f \rangle\}$$

$$DUPS_k = \bigcup \{DUP_k^f : G^f \in S \wedge \text{UNWIND}_P(G^f, k) = \langle G_k^f, DUP_k^f \rangle\}$$

Then, following the simpler representation, we unwind the program P via applying UNWIND_P on duplications $DUPS$. That is,

$$\text{Initially, } P_0 = P, GS_0 = \emptyset, DUPS_0 = \{G^{\text{main}}\}$$

$$\langle GS_k, DUPS_k \rangle = \text{UNWIND}_P(DUPS_{k-1}, k)$$

$$P_k = (P_{k-1} \setminus GS_{k-1}) \cup GS_k \cup DUPS_k$$

As defined above, $DUPS_k$ represents all those duplications of functions in k -th iteration. In k -th iteration, we only unwind functions in $DUPS_{k-1}$ because they are new functions and must not involve in any recursion. As a result, the unwound program P_k could preserve the behavior of original program P .

5.2 Under-approximation

In this section, we first introduce how to under-approximate a given function f with $\underline{G}^f = \text{UNDER}(G^f)$. The produced CFG \underline{G}^f should not contain any recursive calls, and all computations in \underline{G}^f should be valid in original CFG G^f . Further, we explain how we use UNDER combined with two unwinding methods to derive under-approximation of the unwound program P_k .

5.2.1 Approximating Functions

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 call in G with the command

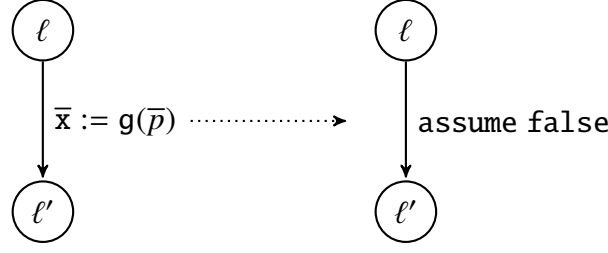


Figure 5.2: Under-approximation

`assume false` (Figure 5.2). That is,

$$\underline{\text{cmd}}^f(\ell, \ell') = \begin{cases} \text{assume false} & \text{if } \text{cmd}^f(\ell, \ell') \text{ contains function calls} \\ \text{cmd}^f(\ell, \ell') & \text{otherwise} \end{cases}$$

Proposition 5.4 *Let G^f be a control flow graph. P and Q are logic formulae with free variables over program variables of G^f . If $\llbracket P \rrbracket G^f \llbracket Q \rrbracket$, then $\llbracket P \rrbracket \underline{G}^f \llbracket Q \rrbracket$.*

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 .

5.2.2 Approximating Programs from Function Inlining Method

Following how we do unwinding in § 5.1.1, we can under-approximate the program through simply computing under-approximation of `main` function as below.

$$\underline{P}_k = \{\underline{G}_k^{\text{main}}\}$$

$\underline{G}_k^{\text{main}}$ doesn't contain any function call, so it is unnecessary to copy other functions from P_k . Thus, the unwound program \underline{P}_k can contain only `main` for less memory usage.

5.2.3 Approximating Programs from Function Duplicating Method

Observing the derivation of P_k in § 5.1.2, we know that all function duplications in $DUPS_i$ for $0 \leq i < k$ do not contain any recursive calls as they were unwound in $(i + 1)$ -iteration.

With this hint, we only have to consider functions in $DUPS_k$ which may contain calls to recursive functions, and we directly under-approximate all functions in $DUPS_k$.

$$\underline{P}_k = (P_k \setminus DUPS_k) \cup \{\underline{G}^{\mathbf{f}_k} : G^{\mathbf{f}_k} \in DUPS_k\}$$

5.3 Computing Summaries

In order to help extract summaries from the inductive invariant, $\text{MarkLocs}(P_k)$ annotates in P_k the *outermost* pair of the entry and exit locations s_i and e_i of each unwound function call to \mathbf{f} with an additional superscript \mathbf{f} , i.e., $s_i^{\mathbf{f}}$ and $e_i^{\mathbf{f}}$. Outermost is defined by *dominance* relation over program locations(nodes). A location ℓ *strictly dominates* ℓ' ($\ell \text{ dom } \ell'$) if and only if every execution path from s to ℓ' must go through ℓ . Similarly, ℓ *post-dominates* ℓ' ($\ell \text{ post-dom } \ell'$) if and only if every execution path from ℓ' to e must go through ℓ . We can thereby define that $\langle s_i, e_i \rangle$ *encloses* $\langle s_j, e_j \rangle$ if only if $s_i \text{ dom } s_j$ and $e_i \text{ post-dom } e_j$, and we can then find outermost pair with this definition that this pair is not enclosed by any other pair through Algorithm 2. Note that $\text{MarkLocs}(P_k)$ is designed to handle unwound programs from both unwinding methods, and it does not provide any output but modify program P_k directly.

Input: P_k : a program

if P_k *is from* UNWIND **then**

foreach $s_i, e_i \in V$ of G^{main} **do**

$\mathbf{f} := \text{GETFUNCTION}(i)$;

if $\neg(\exists j. \mathbf{f} = \text{GETFUNCTION}(j) \wedge s_j \text{ dom } s_i \wedge e_j \text{ post-dom } e_i)$ **then**

 Change s_i, e_i to $s_i^{\mathbf{f}}, e_i^{\mathbf{f}}$;

return;

if P_k *is from* UNWIND_P **then**

foreach function $G^{\mathbf{f}_i}$ in the program P_k **do**

if $\neg(\exists j < i \wedge s_j^{\mathbf{f}}, e_j^{\mathbf{f}} \in V \text{ of } G^{\mathbf{f}_j})$ **then**

 Change s_i, e_i to $s_i^{\mathbf{f}}, e_i^{\mathbf{f}}$;

return;

Algorithm 2: $\text{MarkLocs}(P_k)$

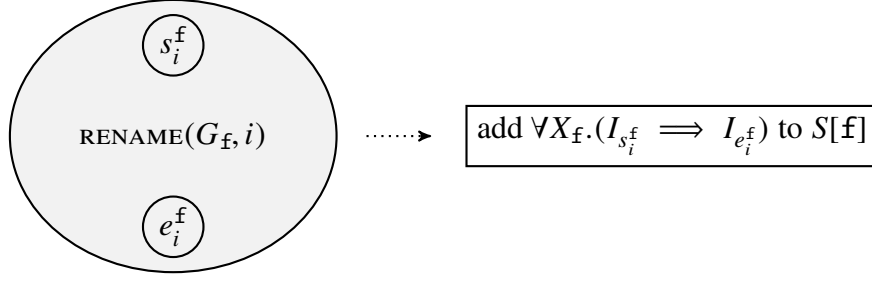


Figure 5.3: Updating a Summary

Now, let the CFG for the main function $\underline{G}_k^{\text{main}} = \langle V, E, \underline{\text{cmd}}^{\text{main}}, \bar{\mathbf{u}}^{\text{main}}, \bar{\mathbf{r}}^{\text{main}}, s, e \rangle$. Function $\text{ComputeSummary}(\mathbf{P}_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$ extracts summaries from the inductive invariant $\Pi(\underline{G}_k^{\text{main}}, \text{true}) = \{I_\ell : \ell \in V\}$ (Algorithm 3).

Input: \mathbf{P}_k : a program; $\{I_\ell : \ell \in V\}$: an inductive invariant of $\underline{G}_k^{\text{main}}$

Output: $S[\bullet]$: function summaries

$\text{MarkLocs}(\mathbf{P}_k)$;

foreach *function \mathbf{f} in the program \mathbf{P}_k* **do**

$S[\mathbf{f}] := \text{true}$;

foreach *pair of locations $\langle s_i^f, e_i^f \rangle \in V \times V$* **do**

if $I_{s_i^f}$ *contains return variables of \mathbf{f}* **then**

$S[\mathbf{f}] := S[\mathbf{f}] \wedge \forall X_f. I_{e_i^f}$

else

$S[\mathbf{f}] := S[\mathbf{f}] \wedge \forall X_f. (I_{s_i^f} \implies I_{e_i^f})$

return $S[\bullet]$;

Algorithm 3: $\text{ComputeSummary}(\mathbf{P}_k, \Pi(\underline{G}_k^{\text{main}}, \text{true}))$

For each function \mathbf{f} in the program \mathbf{P}_k , we first initialize its summary $S[\mathbf{f}]$ to true . The set X_f contains all variables appearing in $\underline{G}_k^{\text{main}}$ except the set of formal parameters and return variables of \mathbf{f} . For each pair 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 contains return variables of \mathbf{f} , we update $S[\mathbf{f}]$ to the formula $S[\mathbf{f}] \wedge \forall X_f. I_{e_i^f}$. Otherwise, we update it to a less restricted version $S[\mathbf{f}] \wedge \forall X_f. (I_{s_i^f} \implies I_{e_i^f})$ (Figure 5.3).

Proposition 5.5 *Let Q be a formula over all variables in $\underline{G}_k^{\text{main}}$ except $\bar{\mathbf{r}}^f$. We have $\llbracket Q \rrbracket \bar{\mathbf{r}}^f := \mathbf{f}(\bar{\mathbf{u}}^f) \llbracket Q \rrbracket$.*

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 3); 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 5.6 *Given 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 $(\text{true}) \bar{r}^f := f(\bar{u}^f) (S[f])$ holds, then $(I_{s_i^f}) \bar{r}^f := f(\bar{u}^f) (I_{e_i^f})$ 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

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

$$\frac{(\text{true}) \bar{r}^f := f(\bar{u}^f) (I_{e_i^f})}{(I_{s_i^f}) \bar{r}^f := f(\bar{u}^f) (I_{e_i^f})} \text{Precondition Strengthening}$$

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

$$\text{Prop. 5.5} \frac{\frac{(\text{true}) \bar{r}^f := f(\bar{u}^f) (S[f])}{(\text{true}) \bar{r}^f := f(\bar{u}^f) (\forall X_f. (I_{s_k^f} \implies I_{e_k^f}))} \quad \frac{(\text{true}) \bar{r}^f := f(\bar{u}^f) (I_{s_k^f} \implies I_{e_k^f})}{(I_{s_k^f}) \bar{r}^f := f(\bar{u}^f) (I_{s_k^f} \implies I_{e_k^f})}}{(I_{s_k^f}) \bar{r}^f := f(\bar{u}^f) (I_{e_k^f})}$$

5.4 Checking Summaries

Here we explain how $\text{CheckSummary}(P_k, S[\bullet])$ is achieved, 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

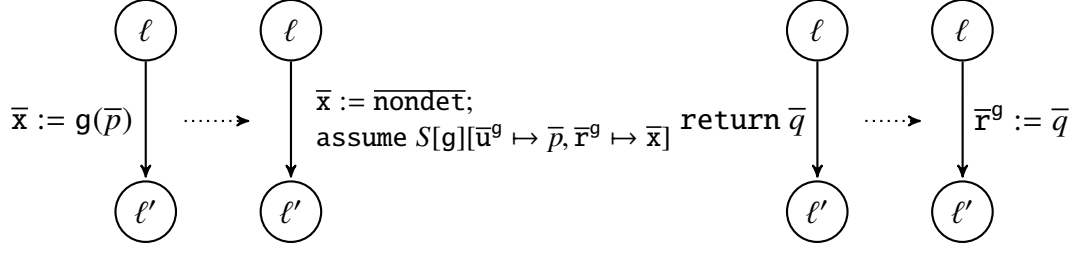


Figure 5.4: Instantiating a Summary

summary $S[f]$ for f specifies the relation between the formal parameters and return values of f , we define another control flow graph $\hat{G}_{k,S}^f = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$ where

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

The control flow graph $\hat{G}_{k,S}^f$ replaces every function call in G_k^f by instantiating a function summary (Figure 5.4). Using the Hoare Logic proof rule for recursive functions [19], we have the following proposition:

Proposition 5.7 *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 $\hat{G}_{k,S}^g = \langle V, E, \text{cmd}^f, \bar{u}^f, \bar{r}^f, s, e \rangle$ 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

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

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  $G_k^f \in P_k$  do
    | if BASICANALYZER( $\tilde{G}_{k,S}^f$ )  $\neq$  Pass then
    | | return false
return true;

```

Algorithm 4: CheckSummary(P_k, S)

Corollary 5.8 *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 BASICANALYZER($\tilde{G}_{k,S}^g$) returns Pass for every function g in P , then $(\text{true}) \bar{r}^f := f(\bar{u}^f) \ (S[f])$.*

5.5 Correctness

We are ready to sketch the proof of Theorem 5.1. In the case that the program P passed our analysis, and Algorithm 1 returns $\text{Pass}(\Pi(\underline{G}_k^{\text{main}}, \text{true}))$ and $S[\bullet]$ on the input control flow graph $G^{\text{main}} = \langle V, E, \text{cmd}^{\text{main}}, \bar{u}^{\text{main}}, \bar{r}^{\text{main}}, s, e \rangle$, the assertions in original program P should hold.

Let $\underline{G}_k^{\text{main}} = \langle \underline{V}_k, \underline{E}_k, \underline{\text{cmd}}_k^{\text{main}}, \bar{u}^{\text{main}}, \bar{r}^{\text{main}}, s, e \rangle$ and $\Pi(\underline{G}_k^{\text{main}}, \text{true}) = \{\underline{I}_\ell : \ell \in \underline{V}_k\}$. By the definition of inductive invariants, we have $(\underline{I}_\ell) \ \underline{\text{cmd}}_k^{\text{main}}(\ell, \ell') \ (\underline{I}_{\ell'})$ for every $(\ell, \ell') \in \underline{E}_k$. Moreover, $V \subseteq \underline{V}_k$ since $\underline{G}_k^{\text{main}}$ is obtained by unwinding G^{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^{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 $(\underline{I}_\ell) \ \text{cmd}^{\text{main}}(\ell, \ell') \ (\underline{I}_{\ell'})$ 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

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

for every $(\ell, \ell') \in \hat{E}$. By the inductiveness of $\Pi(\underline{G}_k^{\text{main}}, \text{true})$, we have $\llbracket \underline{I}_\ell \rrbracket \bar{\mathbf{u}}^f := \bar{p} \llbracket \underline{I}_{s_k^f} \rrbracket$ and $\llbracket \underline{I}_{e_k^f} \rrbracket \bar{\mathbf{x}} := \bar{\mathbf{r}}^f \llbracket \underline{I}_{\ell'} \rrbracket$. Moreover, $\llbracket \underline{I}_{s_k^f} \rrbracket \bar{\mathbf{r}}^f := \mathbf{f}(\bar{\mathbf{u}}^f) \llbracket \underline{I}_{e_k^f} \rrbracket$ by Proposition 5.6 and 5.7.

Therefore

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

Chapter 6

Experiments

6.1 Implementation

A prototype of our approach, CPAREC, is implemented with CPACHECKER 1.2.11-svcomp14b as the underlying BASICANALYZER. The overall software architecture and execution flow is illustrated in Figure 6.1. For all program transformations mentioned in Chapter 5, we benefit from the well-known front-end, C Intermediate Language (CIL) [27], [28], to apply all required transformations on C programs. In addition, because CPACHECKER does not support universal quantifiers in the expression of an `assume` command, we used REDLOG [29], [30] for quantifier elimination.

CPAREC is available at <https://github.com/fmlab-iis/cparec>. The experiments in this paper is based on version v0.1-alpha. The simplest way to execute CPAREC is to first download the binary from the web-site. To setup the environment in Ubuntu 12.04 64-bit, JAVA Runtime, Python 2.7, the Python Networkx package, and the Python PyGraphviz package are required. Run following commands to install above packages in Ubuntu 12.04 64-bit:

```
sudo apt-get install openjdk-7-jre
sudo apt-get install python python-networkx python-pygraphviz
```

To process a benchmark example `program.c`, one should use the following script:

```
python <path_to_cparec>/cparec/main.py program.c
```

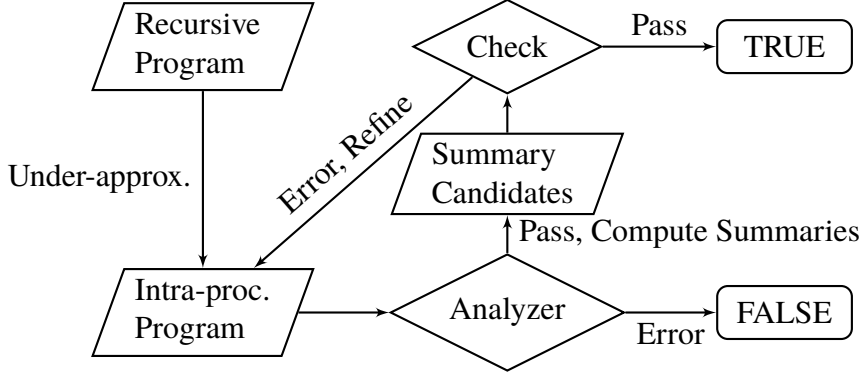


Figure 6.1: The Execution Flow of CPAREC

No further parameters are needed. CPAREC will print the verification result to the console.

6.2 Experiments

To evaluate our tool, we performed experiments with all the benchmarks from the **recursive** category in the 3rd Competition on Software Verification (SV-COMP 2014) [3] and followed the rules and the scoring scheme (shown in Table 6.1) 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 2.7.2¹ [7], CBMC 4.5-sv-comp-2014 [15] with a wrapper cbmc-wrapper.sh², ULTIMATE AUTOMIZER [16], and ULTIMATE KOJAK [17]. 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.

¹ We use the arguments **-alias empty -enable-recursion -noprofile -cref -sv-comp -lattice -include-lattice symb -nosserr** with BLAST.

² The wrapper cbmc-wrapper.sh is provided by CBMC 4.5-sv-comp-2014, which is a special version for SV-COMP 2014.

Table 6.1: Scoring scheme 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

The results are summarized in Table 6.2 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 bound³, 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 6.2 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.

In addition to the experiments, we also participated SV-COMP 2015 [5] and competed with other 8 teams under the **recursive** category. The scoring scheme in SV-COMP

³ This was confirmed in a private communication with the developers of CBMC.

Table 6.2: 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
	<i>k</i>	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

Table 6.3: Scoring scheme in SV-COMP 2015.

Points	Program Correctness	Reported Result
0	TRUE or FALSE	UNKNOWN (due to timeout or exceptions)
+1	FALSE	FALSE
-6	TRUE	FALSE
+2	TRUE	TRUE
-12	FALSE	TRUE

2015 (Table 6.3) increases penalty points on both false positive and false negative results. Therefore, the strategy of CBMC didn't do the trick this year. Table 6.4 provides partial results containing the final scores for the top 5 tools. The data are quoted from the competition report⁴. Our tool, CPA_{REC}, won the Third place among all 9 competitors.

One exciting fact is that CPA_{CHECKER} participated in RECURSIVE this year with a dedicated extension to support recursion [31]. However, our tool performed slightly better than this version of CPA_{CHECKER}, and, thereby, prevented CPA_{CHECKER} from winning their 9th medal in SV-COMP 2015.

Another notable result is that, if we observe the accumulated score over time (Figure 6.2), our tool acquired the same score with less time compared with SMACK and ULTIMATE AUTOMIZER. That means that we can solve easy cases faster, and a possible reason is that we limit our underlying analyzer to use linear integer arithmetic logic as the program semantic.

Finally, there is a considerable portion of TRUE cases our tool cannot solve but SMACK and ULTIMATE AUTOMIZER can. The reason is the same that our program semantic is not expressive enough to conclude the safeness of the program.

⁴ Available at <http://sv-comp.sosy-lab.org/2015/results/>
Select **recursive** category for the complete table.

Table 6.4: Partial competition results of the **recursive** category of the 2015 Competition on Software Verification.

Results	CPAREC	ULTIMATE AUTOMIZER	ULTIMATE KOJAK	SMACK	CPACHECKER 1.3.10-svcomp15
correct	12	16	8	23	11
false negative	0	0	0	1	0
false positive	0	0	0	0	0
unknown	12	8	16	0	13
score	18	25	10	27	16

Figure 6.2: Accumulated Score over Time

Chapter 7

Conclusion

The number of iterations is perhaps the most important factor in our recursion analysis technique (Table 6.2) as it would determine how many times of unwinding are applied. We find that CPACHECKER performs poorly when checking programs that is unwound many times. 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 4) 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.

Another improvement on our algorithm is on selecting locations for extracting inductive invariants. In Algorithm 2, we select only outermost pairs of locations for calls to the same function. This is based on the observation that the unwound bodies of these function calls contain more execution paths, and hence their behaviors should be closer to the original function. However, the extracted invariants in s_i may be too precise to those certain function calls and result in too coarse summary candidates constructed by

implication connective, and consequently the candidates can not pass CheckSummary due to inner function calls are not properly approximated. Therefore, heuristics that select some locations of inner function calls may help compute summary candidates with better quality.

Further enhancement and application of our work would be supporting recursive data structures and verifying operations on the data structure. Common recursive functions in real world C program mostly are simple operations on recursive data structures, such as list, tree, graph, etc., but our prototype currently only verifies integer programs. If our approach is adapted to handle data structures, a dedicated logic is required for describing data structure as well as semantic for the operations, and a BASICANALYZER providing inductive invariants in such logic surely is necessary. Our primitive study reveals several difficulties on applying our approach with separation logic [32] and the THOR analyzer [33].

One major obstacle comes from ComputeSummary and CheckSummary. In our method, implication connective and universal quantification are used to derive summary. These operators are intuitive in propositional and Presburger arithmetic logic. By contrast, there is no such operator on separation logic. Consequently, we tried to devise different methods for compute and check summaries in order to avoid the usage of the operators. A possible workaround for this problem is to use multiple formulae instead of one single formula to represent one function summary.

Another problem arises from the expression allowed in `assume` and `assert` command. Our program model does not introduce a separated *assertion language* for specifying preconditions and postconditions, and this is to comply with the assertions in C language. The drawback of this choice is that the expressiveness and grammar of allowed boolean expression in program is limited, and hence not necessarily consistent with the logic in BASICANALYZER. To precisely describe a recursive data structure, we tried to introduce annotations recognized by BASICANALYZER into C programs. In our study, we actually used the assertion language defined by THOR to annotate C programs, and much more efforts are needed to understand the underlying analyzer, not to mention the transformation on the formulae.

Finally, reducing program features via program transformation techniques is the core concept of our work. We believe this idea can be applied to deal with other kinds of program features, such as pointers, unbounded integral types, procedure calls through dynamic look up, absent code due to external libraries, etc. Program analysis tools can rely on one efficient and simple core `BASICANALYZER` and modular components doing transformation for different program features. The development tasks for the tools could be easily divided and dispatched, and optimization for different components could be independent and loosely coupled. Overall, we believe the concept could speed up the process of research as well as implementation on program analysis.

Bibliography

- [1] E. M. Clarke, H. Jain, and N. Sinha, “Grand challenge: Model check software,” in *Proc. the NATO Advanced Research Workshop on Verification of Infinite State Systems with Applications to Security (VISSAS 2005)*, Timișoara, România, 2005, pp. 55–68.
- [2] D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *Proc. CAV’11*, Snowbird, UT, USA, 2011, pp. 184–190.
- [3] D. Beyer, “Status report on software verification - (competition summary SV-COMP 2014),” in *Proc. TACAS’14*, Grenoble, France, 2014, pp. 373–388. [Online]. Available: <http://sv-comp.sosy-lab.org/2014/>.
- [4] Y. Chen, C. Hsieh, M. Tsai, B. Wang, and F. Wang, “Verifying recursive programs using intraprocedural analyzers,” in *Proc. SAS’14*, Munich, Germany, 2014, pp. 118–133.
- [5] D. Beyer, “Software verification and verifiable witnesses - (report on SV-COMP 2015),” in *Proc. TACAS’15*, London, UK, 2015, pp. 401–416. [Online]. Available: <http://sv-comp.sosy-lab.org/2015/>.
- [6] Y. Chen, C. Hsieh, M. Tsai, B. Wang, and F. Wang, “Cparec: Verifying recursive programs via source-to-source program transformation - (competition contribution),” in *Proc. TACAS’15*, London, UK, 2015, pp. 426–428.
- [7] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker BLAST,” *Int. J. Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, Sep. 2007.

- [8] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, “Ufo: A framework for abstraction- and interpolation-based software verification,” in *Proc. CAV’12*, Berkeley, CA, USA, 2012, pp. 672–678.
- [9] T. W. Reps, S. Horwitz, and S. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *Proc. POPL’95*, San Francisco, CA, USA, 1995, pp. 49–61.
- [10] T. Ball and S. K. Rajamani, “The SLAM toolkit,” in *Proc. CAV’01*, Paris, France, 2001, pp. 260–264.
- [11] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The ASTREÉ analyzer,” in *Proc. ESOP’05*, Edinburgh, UK, 2005, pp. 21–30.
- [12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, “Frama-c: A software analysis perspective,” in *Proc. SEFM’12*, Thessaloniki, Greece, 2012, pp. 233–247.
- [13] (2015). Coverity, Coverity, Inc., [Online]. Available: <http://www.coverity.com/>.
- [14] (2015). Polyspace, The MathWorks, Inc., [Online]. Available: <http://www.mathworks.com/products/polyspace/>.
- [15] E. M. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Proc. TACAS’04*, Barcelona, Spain, 2004, pp. 168–176.
- [16] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski, “Ultimate Automizer with SMTInterpol - (competition contribution),” in *Proc. TACAS’13*, Rome, Italy, 2013, pp. 641–643.
- [17] E. Ermis, A. Nutz, D. Dietsch, J. Hoenicke, and A. Podelski, “Ultimate Kojak - (competition contribution),” in *Proc. TACAS’14*, Grenoble, France, 2014, pp. 421–423.
- [18] A. Albarghouthi, A. Gurfinkel, and M. Chechik, “WHALE: An interpolation-based algorithm for inter-procedural verification,” in *Proc. VMCAI’12*, Philadelphia, PA, USA, 2012, pp. 39–55.

- [19] D. von Oheimb, “Hoare logic for mutual recursion and local variables,” in *Proc. FSTTCS*, Chennai, India, 1999, pp. 168–180.
- [20] A. Lal and T. W. Reps, “Reducing concurrent analysis under a context bound to sequential analysis,” in *Proc. CAV’08*, Princeton, NJ, USA, 2008, pp. 37–51.
- [21] ———, “Reducing concurrent analysis under a context bound to sequential analysis,” *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, Aug. 2009.
- [22] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag, 1999.
- [23] T. Ball and S. K. Rajamani, “Bebop: A symbolic model checker for boolean programs,” in *Proc. SPIN*, Stanford, CA, USA, 2000, pp. 113–130.
- [24] Z. Manna and A. Pnueli, “Formalization of properties of functional programs,” *J. ACM*, vol. 17, no. 3, pp. 555–569, Jul. 1970.
- [25] H. R. Nielson and F. Nielson, *Semantics with Applications: AN Appetizer*, ser. Undergraduate Topics in Computer Science. London, UK: Springer-Verlag, 2007.
- [26] G. Tan and A. W. Appel, “A compositional logic for control flow,” in *Proc. VM-CAI’06*, Charleston, SC, USA, 2006, pp. 80–94.
- [27] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of C programs,” in *Proc. CC’02*, Grenoble, France, 2002, pp. 213–228.
- [28] (2015). C Intermediate Language, [Online]. Available: <http://kerneis.github.io/cil/>.
- [29] A. Dolzmann and T. Sturm, “Redlog: Computer algebra meets computer logic,” *SIGSAM Bull.*, vol. 31, no. 2, pp. 2–9, Jun. 1997.
- [30] (2015). Redlog, [Online]. Available: <http://www.redlog.eu/>.
- [31] M. Dangl, S. Löwe, and P. Wendler, “CPAchecker with support for recursive programs and floating-point arithmetic - (competition contribution),” in *Proc. TACAS’15*, London, UK, 2015, pp. 423–425.

- [32] J. C. Reynolds, “Separation logic: A logic for shared mutable data structures,” in *Proc. LICS’02*, Copenhagen, Denmark, 2002, pp. 55–74.
- [33] S. Magill, M. Tsai, P. Lee, and Y. Tsay, “Thor: A tool for reasoning about shape and arithmetic,” in *Proc. CAV’08*, Princeton, NJ, USA, 2008, pp. 428–432.
- [34] *Proc. 20th Int. Conf. Computer Aided Verification (CAV’08)*, Princeton, NJ, USA, 2008.
- [35] *Proc. 21st Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS’15)*, London, UK, 2015.
- [36] *Proc. 20th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS’14)*, Grenoble, France, 2014.