COUNTEREXAMPLE-GUIDED EQUIVALENCE CHECKING

SHUBHANI

AMAR NATH & SHASHI KHOSLA SCHOOL OF INFORMATION TECHNOLOGY
INDIAN INSTITUTE OF TECHNOLOGY DELHI
FEBRUARY 2023

COUNTEREXAMPLE-GUIDED EQUIVALENCE CHECKING

by

SHUBHANI

Amar Nath & Shashi Khosla School of Information Technology

Submitted

in fulfillment of the requirements of the degree of Doctor of Philosophy

to the

Indian Institute of Technology Delhi
FEBRUARY 2023

Certificate

This is to certify that the thesis titled **Counterexample-Guided Equivalence Checking** being submitted by **Ms. Shubhani** for the award of **Doctor of Philosophy** in **Amar Nath & Shashi Khosla School of Information Technology** is a record of bona fide work carried out by her under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Sorav Bansal
Associate Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India - 110016

Acknowledgements

The successful completion of this dissertation would not have been possible without the help and support of many people. I wish to acknowledge all these people who have in their own ways played a significant role in my Ph.D. journey. First, I would like to thank my advisor, **Prof. Sorav Bansal**, for showing faith in my capability and giving me the opportunity to pursue a Ph.D. I was a beginner in the area of my research at the time of joining and as a mentor, he provided immense support in my journey of learning the area and choosing the initial problem statements that have helped me gain confidence. I wish to thank him for helping me grow from a beginner to an independent researcher by allowing me to explore my own ideas while providing guidance and support. I also thank him for the constant feedback he provided on my research writing and presentations that have helped me refine the skills necessary to succeed in research and academia. Most importantly, he provided a healthy work environment that helped me in balancing my personal and professional life smoothly.

My sincere thanks to the research committee members **Prof. Sanjiva Prasad, Dr. Rahul Sharma, and Prof. Subodh Sharma** who were always approachable and have provided useful feedback on my work on a regular basis.

There were several fellow students that helped me in various ways during my graduate studies. I would like to thank Abhishek Rose for collaborating with me on this work. This work would not have been completed without him. Both his technical contributions and his insights are an invaluable and inseparable part of this work. I would also like to thank Dr. Manjeet Dahiya for helping me during my initial years as a graduate student. Finally, I would like to thank Aseem Saxena who has contributed to different aspects of this work.

I take this opportunity to express my earnest gratitude to Dr. Rajesh Kedia for his valuable guidance in making some of the hardest and important decisions of my Ph.D. Our lab and office staff – Ms. Vandana, Mr. Rajesh, and Mr. Suresh had been very humble throughout and helped avoiding many administrative hassles. I would like to thank my dear friends Geeta, Aruna Bansal, Himanshu Gandhi, Divya Praneetha, Sanjana Singh and Divyanjali for making this journey much more enjoyable and easy for me by providing advice, and fun distractions along the whole way.

On a personal note, during my PhD, me and my husband were blessed with our son Shivin – I am thankful to God for his grace. Even imagining the thought of starting a Ph.D. after marriage and pursuing it with a child is very challenging. I could start this journey and was able to survive it through the various ups and downs only because of endless support, encouragement and a lot of understanding and adjustments by my husband, Ashwani. This thesis is a result of his patience, dedication and endless support.

I am also grateful to my mother-in-law, father-in-law and brother-in-law Atul for understanding my commitment towards my work and supporting me in fulfilling it in every possible way. I am fortunate to have the best teachers for life as my parents. My mother (Ms. Seema Gupta) taught me the importance of hard work and consistency in life to achieve goals and my father (Late Mr. Vikas Gupta) instilled selflessness and gratitude in me. My younger brother Vibhor has always been my stress buster and my support system. It was my father's dream for me to pursue higher studies and I strongly believe that I could successfully complete this only because of his blessings. I dedicate this thesis to my father and my family.

Shubhani

Abstract

Software is rapidly becoming a key component for an increasing number of critical applications ranging from automotive, aerospace, banking, healthcare, and many more. Formal software verification, which involves verifying the software properties for all possible inputs, has thus become crucial. But the absence of the formal guarantees for the compiler that is used to translate the software to the final executable code limits the guarantees provided by the formal source code verification.

Equivalence checking involves verifying the functional equivalence between a program specification and its implementation. An equivalence checker can be used in a translation validation approach to provide formal correctness guarantees for the executable code generated by the compiler. A black-box equivalence checker takes a program-pair as input and makes minimal assumptions on the exact nature of transformations performed from one program to another.

The general equivalence checking problem is undecidable and is very challenging in the translation validation context due to the potentially large syntactic gap between the source and assembly representation and the long and complex nature of transformations/optimizations performed by the modern optimizing compilers. These optimizations result in significant structural differences

between the input source code and the optimized output code. *This work proposes two algorithms that help make significant progress in the space of robust translation validation.*

The first algorithm efficiently finds the correlation between program transitions for structurally significantly different (but bisimilar) program-pairs and the second algorithm finds a general class of relations between state elements of programs that have significant syntactic gap across them. Both these algorithms are static and do not rely on execution traces. Instead, they purely rely on the concrete models (aka counterexamples) returned by the SMT solvers.

The third contribution of this work is *the first black-box equivalence checking tool that can automatically compute equivalence across the unoptimized intermediate representation (IR) of a program and its optimized x86 assembly implementation generated either by an optimizing compiler or developed by a human programmer.* We use a custom IR representation that resembles LLVM IR to specify the input program. The long and rich pipeline of transformations from the unoptimized IR to the optimized x86 assembly includes optimizations like loop unrolling, peeling, unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, and many more.

सार

ऑटोमोटिव, एयरोस्पेस, बैंकिंग, हेल्थकेयर, और कई अन्य महत्वपूर्ण अनुप्रयोगों की बढ़ती संख्या के लिए सॉफ्टवेयर तेजी से एक प्रमुख घटक बनता जा रहा है। औपचारिक सॉफ्टवेयर सत्यापन, जिसमें सभी संभावित इनपुट के लिए सॉफ़्टवेयर गुणों की पुष्टि करना शामिल है, इस प्रकार से महत्वपूर्ण हो गया है। लेकिन अनुवाद करने के लिए उपयोग किए जाने वाले संकलक के लिए औपचारिक गारंटी का अभाव, अंतिम निष्पादन योग्य कोड के लिए स्रोत कोड औपचारिक सत्यापन द्वारा प्रदान की गई गारंटी को सीमित करता है।

तुल्यता जाँच में प्रोग्राम विनिर्देश और इसका कार्यान्वयन के बीच कार्यात्मक तुल्यता की पुष्टि करना शामिल है। एक अनुवाद सत्यापन दृष्टिकोण में एक तुल्यता परीक्षक का उपयोग किया जा सकता है संकलक द्वारा उत्पन्न निष्पादन योग्य कोड के लिए औपचारिक शुद्धता की गारंटी प्रदान करने के लिए। ए ब्लैक-बॉक्स तुल्यता परीक्षक एक प्रोग्राम-जोड़ी को इनपुट के रूप में लेता है और एक प्रोग्राम से दूसरे प्रोग्राम में किए गए रूपांतरणों की सटीक प्रकृति पर न्यूनतम धारणा बनाता है।

सामान्य तुल्यता जाँच समस्या अनिर्णीत है और अनुवाद सत्यापन संदर्भ में बहुत चुनौतीपूर्ण है स्रोत और असेंबली के बीच संभावित रूप से बड़े सिंटैक्टिक अंतर के कारण और आधुनिक अनुकूलन संकलक द्वारा किए गए परिवर्तनों / अनुकूलन की लंबी और जटिल प्रकृति के कारण। इन अनुकूलन के परिणामस्वरूप महत्वपूर्ण संरचनात्मक अंतर होते हैं स्रोत कोड और अनुकूलित आउटपुट कोड के बीच। यह काम दो एल्गोरिदम प्रस्तावित करता है जो मजबूत अनुवाद सत्यापन के क्षेत्र में महत्वपूर्ण प्रगति करने में मदद करते हैं।

पहला एल्गोरिथम संरचनात्मक रूप से प्रोग्राम ट्रांज़िशन के बीच सहसंबंध को काफी अलग (लेकिन बिस्मिलर) प्रोग्राम-जोड़े के लिए कुशलतापूर्वक पाता है और दूसरा एल्गोरिदम उन प्रोग्रॅम्स के राज्य तत्वों के बीच संबंधों का एक सामान्य वर्ग पाता है जिनके बीच महत्वपूर्ण वाक्यात्मक अंतर है। ये दोनों एल्गोरिदम स्थिर हैं और निष्पादन के निशान पर निर्भर नहीं हैं। इसके बजाय, वे विशुद्ध रूप से भरोसा करते हैं एसएमटी सॉल्वर द्वारा लौटाए गए ठोस मॉडल (उर्फ काउंटरएक्सप्लेम्स) पर।

इस कार्य का तीसरा योगदान पहला ब्लैक-बॉक्स तुल्यता जाँच उपकरण है जो खुद ब खुद एक प्रोग्राम के अनुकूलन-रहित इंटरमीडिएट रिप्रेजेंटेशन और इसके अनुकूलित असेंबली कार्यान्वयन, जो या तो एक अनुकूलन संकलक द्वारा उत्पन्न हो या एक मानव प्रोग्रामर द्वारा विकसित हो, में समतुल्यता की गणना करता है। हम प्रोग्राम निर्दिष्ट करने के लिए एक कस्टम आईआर प्रतिनिधित्व का उपयोग करते हैं जो एलएलवीएम आईआर इनपुट जैसा दिखता है।
अनुकूलन-रहित इंटरमीडिएट रिप्रेजेंटेशन और इसके अनुकूलित असेंबली कार्यान्वयन में परिवर्तनों की लंबी और समृद्ध पाइपलाइन में लूप अनरोलिंग, पीलिंग, अनस्विचिंग, वर्जनिंग, लूप इनवर्जन, वेक्टराइज़ेशन, रजिस्टर अलोकेशन, कोड उत्थापन, शक्ति कमी, मृत कोड उन्मूलन, जैसे अनुकूलन शामिल हैं।

Contents

Abstract **v**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**

1.1. Contributions and Organization 4

2 Preliminaries **7**

2.1. Formal Definition of Equivalence in the context of Translation Validation 7

2.2. Control-Flow Graph Representation 9

2.2.1. Modeling Undefined Behavior 11

2.2.2. Well-formed CFG 13

2.3. Equivalence Checking Through Simulation Relation Construction 14

2.3.1. Product-CFG 14

2.3.2. Incremental Algorithm for Product-CFG Construction 18

2.4. Counterexample 24

3 Counterexample-Guided Correlation Algorithm 27

3.1. Introduction 27

3.2. Related Work and Motivating Examples 29

3.2.1. Motivating Example 1 31

3.2.2. Motivating Example 2 33

3.2.3. Motivating Example 3 35

3.3. *Counter* Algorithm 37

3.3.1. Top-level Procedure for *Counter* Algorithm 37

3.3.2. Incremental Procedure to add a new product-CFG edge 41

3.3.3. Pathset Correlation 43

3.3.4. Algorithm for Enumerating Candidate Pathsets 46

3.3.5. Criterion for Correlating Pathsets 55

3.3.6. Counterexample Propagation 56

3.3.7. Counterexample-Guided Pruning and Ranking 58

3.3.8. Pruning and Ranking Algorithms Through Examples 62

3.3.9. Contrast with SPA Algorithm 65

4 Counterexample-Guided Invariant Inference 67

4.1. Prior work on Invariant Inference Techniques 68

4.1.1. Program Execution Based Techniques. 69

4.1.2. Syntax/Enumeration Based Techniques 71

4.1.3. Constraint-Solving Based Techniques 72

4.2. *Sifer* Algorithm 74

4.2.1. Strongest Inductive Invariant Cover 75

4.2.2. Data-Flow Analysis Framework 78

4.2.3. Characteristics of the Algorithm 83

4.2.4. Comparison with the abstract interpretation based prior work 85

4.2.5. Computation of the SInvCover() 87
4.2.6. Explaining the Sifer algorithm through an example 89

5 Unoptimized-IR-to-Optimized-Assembly Translation Validator 93
5.1. Implementation Details 94
5.1.1. Logical Representation 94
5.1.2. Deterministic CFG Construction 95
5.1.3. Observable actions 96
5.1.4. Memory Model 96
5.1.5. Points-to and other standard data-flow analyses 98
5.1.6. Invariant Inference Grammar 100
5.1.7. Discharging proof obligations 101
5.2. Evaluation 102
5.2.1. Experimental Setup 102
5.2.2. Results 104
5.3. Limitations 113
5.3.1. *Counter* Algorithm 113
5.3.2. *Sifer* Algorithm 114
5.4. Comparison with Prior Translation Validation Tools 116

6 Conclusion 123

Bibliography 127

List of Publications 139

Biography 141

List of Figures

2.1 An example C-language program for dead code elimination (DCE) optimization. . 8

2.2 An example C-language program and its control-flow graph representation. 10

2.3 CFG with UB assumptions for C program shown in fig. 2.2a 12

2.4 An example C-language program, its equivalent (abstracted) assembly language program, their CFG representation and simulation relation across the CFGs represented as a product-CFG and the inductive invariants at the nodes of the product-CFG. 15

2.5 Pseudo code for the incremental product-CFG construction algorithm 19

2.6 A snapshot of the backtracking search tree for the program-pair shown in fig. 2.4 . 21

3.1 An example program-pair taken from TSVC suite such that the product-CFG across them requires an exponential number of paths in an edge. 32

3.2 An example program-pair and the required product-CFG for loop splitting and loop unswitching optimizations. 34

3.3 An example program-pair with multiple loops taken from polybench suite and the required product-CFG across them with different alignment conditions for different loops. 35

3.4 Pseudo code for the top-level procedure of *Counter* algorithm 38

3.5 A snapshot of the backtracking search tree for the program-pair shown in fig. 3.3 . 39

3.6 Pseudo code for the incremental procedure to expand a given partial product-CFG 42

3.7 SP-graph and expanded representation of a 2-unrolled pathset starting from node `a` to node `a`. 45

3.8 Pseudo code for the *(μ, δ)-unrolled pathset* enumeration algorithm used in *Counter*. 48

3.9 Pseudo code for the algorithm to enumerate the pathset in the implementation CFG. 53

3.10 Pseudo code for the algorithm to enumerate the candidate pathsets in the specification CFG 54

3.11 Comparison function used to rank product-CFGs during best-first search. The comparison operators $<, \leq$ for tuples compare lexicographically starting with the first element. 61

4.1 A C program that counts the number of elements in a linked list. 70

4.2 An example C program that initializes an array `a`. 76

4.3 Transfer function and Meet operator for Invariant Inference DFA in table 4.1. `SInvCover()` computes the strongest invariant cover for a set of counterexamples. p_ω represents the concrete execution function for edge ω. γ_n is the counterexample returned by the SMT solver for a `SAT()` query. 81

4.4 SMT solver based algorithm to find the most precise inductive $\mathcal{A}$-*invariant* proposed by the abstract interpretation based prior work [68, 58]. Here, $\mathcal{A}$ denotes the abstract domain. 85

5.1 The grammars used by COUNTER tool for constructing the invariants. 100

5.2 An example C-language program to initialize an array, its abstracted assembly after loop unroll and vectorization, and the product-CFG across the two. 103

5.3 The C code, the optimized assembly (after loop splitting and unswitching) and the product-CFG for an example loop nest from LORE benchmark 108

5.4 The bug in *diet libc* identified using the COUNTER tool. 111

5.5 Example program-pair for which *Counter* algorithm will not be able to construct the required product-CFG. 113

5.6 An example C code and its optimized assembly obtained after store sinking optimization 115

List of Tables

4.1 Data-flow formulation of *Sifer* algorithm for inference of inductive invariants drawn from the input grammar $\mathbb{G}$. 79

5.1 Evaluation results for the COUNTER tool for TSVC benchmark functions and LORE loop nest patterns. 105

5.2 List of passing vectorized TSVC functions. For a function-compiler pair, ✗denotes equivalence check failure and $\otimes$ denotes that the function is not vectorized by the compiler. Here, the prior work refers to the work by Churchill et. al. [11] 107