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

Chapter 1

Introduction

The problem of equivalence checking, i.e. verifying the functional equivalence between a program specification and its implementation has been of much research interest and has several important applications in translation validation [52, 66, 69], program (super)optimization [3, 60, 4, 46, 55, 62], regression verification [65, 26], and many more [38, 42, 67]. The translation validation attempts to automatically generate a proof of equivalence across the transformations (translations) performed by an optimizing compiler. This involves constructing an equivalence checker which after every run of the compiler verifies the functional equivalence between the input source code and the generated executable code. The program synthesis (or superoptimization) involves using program synthesis to automatically generate (or learn) optimizations for a given program specification. The equivalence checker in this application is tasked with determining if the (randomly) proposed optimized program is equivalent to the input program specification.

For most of these applications, soundness of an equivalence checker is critical, i.e., if the equivalence checker determines the programs to be equivalent, then the programs must be guaranteed to have equivalent runtime observable behavior for all legal inputs. An unsound equivalence checker may result in verification of an incorrect translation and is unacceptable for most of the applications mentioned above. On the other hand, because the general problem of equivalence checking is undecidable, completeness may not always be achievable for a sound equivalence checker. So,

it is possible that the equivalence checker is unable to prove the programs equivalent, even if they were behaviorally equivalent. In the context of translation validation, the lack of completeness results in false negatives, i.e. the equivalence checker reports false compilation bugs and for program synthesis/superoptimization this results in missed optimization opportunities. Thus the precision and capability of the equivalence checking applications (including translation validation and program synthesis) rely on the robustness of the equivalence checker used. In this work, we propose algorithms for building a more robust equivalence checker which can compute equivalence for a larger set of program-transformation pairs as compared to the state-of-the-art. Although we use the translation validation for discussing the proposed algorithms, the techniques presented are largely applicable to other equivalence checking applications as well.

The most common approach for equivalence checking involves establishing a bisimulation relation across the given program-pair. The (bi)simulation relation construction algorithm consists of two major steps (1) Correlating program transitions (or paths) across the specification and the implementation programs (2) Identifying inductively-provable relations (or invariants) between variables (state-elements) of the two programs at the endpoints of the correlated transitions [56]. If these correlations and inductive invariants ensure equivalent observable behavior (e.g., identical sequence of I/O events, identical return values of the program and the non-temporary memory state at the exit), then we have obtained a proof (or witness) of equivalence (and bisimilarity). This proof, involving correlations and invariants, can be represented either as a (bi)simulation relation [56, 48, 52] or as a product program [74], both of which are equivalent representations.

In the context of translation validation, the program specification represents the language-level semantics of the program, e.g., the C language semantics of a C program, and the implementation models the semantics of the low-level assembly opcodes that run directly on the hardware, e.g., the semantics of the x86 assembly instructions. Thus, an ideal translation validator in this setting should be able to automatically compute equivalence across the full pipeline of compiler transformations from the C-language source code to the x86 assembly program implementation. This is very challenging because

(a) The large and complex nature of transformations/optimizations performed by the modern optimizing compilers result in significant structural differences between the input source

code and the optimized output code. Robust and exhaustive algorithms (with potentially exponential search space) are required to correlate the transitions across the input program and the optimized output program.

(b) Simple invariants (like equality or inequality relation between two program variables) do not suffice for the large syntactic gap between the source and assembly representation. More expressive invariants (like affine invariants) are required to be inferred across the state-elements of the two programs.

To tackle this challenging equivalence checking problem at the scale of real world programs, prior work has usually simplified the problem by one of the following approaches or problem settings

- Using the same abstraction level for both the specification and implementation programs. For example, both the programs could be in high-level IR representation or both the programs may be in low-level assembly representation. This may simplify the invariants required to prove equivalence and may benefit from syntax based heuristics for invariant inference. The shortcoming of using IR as both specification and implementation [52, 69, 25, 75, 44] is that it misses some of the most involved transformations (like instruction selection) that are critical for verification. Using assembly as both specification and implementation [63, 14, 60, 11] may result in more false equivalence failures because information to prove a transformation correct may not be available in the assembly level specification (as compared to source code or IR level specification).
- Modifying the compiler source code to produce a witnesses (or proof) [51, 35] during compilation itself. This places extra burden on the compiler developers and increases the compiler complexity. Further, this approach for verification can not be used for constructing an equivalence checker for the application of program synthesis. An equivalence checker for program synthesis/superoptimization should be agnostic to the exact nature of transformations performed while proposing the (randomly chosen) optimized program.
- Computing equivalence only across selected components/transformations of the compiler [25, 44] such that the resulting program after the transformation is structurally similar to the input program. This reduces the equivalence checking problem to only invariant inference problem as the structurally similar program-pair can be correlated trivially.

- Executing both the programs on real inputs and using the execution output for establishing the correlation and inferring the required invariants. Although these program execution-based techniques [63, 23, 11, 53] are effective, their precision/completeness depends on the coverage of the execution traces. Also, generating these execution traces can have practical limitations for programs at different level of abstractions; this has not been demonstrated yet in any prior work.
- Correlating the whole program in a single step mostly using a common alignment condition [63, 11]. This simplifies the simulation relation construction problem by partitioning it into two completely disjoint steps for correlation and invariant inference. The invariant inference algorithm in this setting is usually performed after the correlation for the complete program has been established. Sophisticated invariant inference techniques (like [13, 26]) can now be used for relating the state-elements for the complete correlated program. The biggest shortcoming of this approach is that different fragments (or loops) in a program may require different alignment conditions for correlation and this approach cannot handle such programs by correlating the whole program in a single step using a common alignment condition. Prior work has demonstrated these common alignment condition based correlation on mostly single-loop source programs only.

1.1 Contributions and Organization

This thesis makes the following high-level contributions to address the shortcomings of prior work:

- The first contribution is a *counterexample-guided correlation* algorithm called *Counter*. Counterexamples represent the concrete models returned by an SMT solver for queries that are not provable. Unlike most prior work, which correlates the whole program in a single step, *Counter* is an incremental algorithm for identifying the correlations and can handle programs with multiple loops which may have different alignment constraints. At each incremental step, the proposed algorithm uses a *counterexample-guided best-first search* strategy to choose the new correlated transition pair from an exponential search space and performs the invariant inference across the chosen transition pair. The proposed best-first

search strategy involves (a) counterexample execution based pruning of infeasible correlations to reduce the search space (b) choosing the most promising candidate correlation using a ranking procedure based on the invariants inferred so far. Using the proposed best-first search strategy, *Counter* can efficiently identify the required correlations across structurally significantly different (but bisimilar) program-pairs.

Chapter 3 discusses the proposed *Counter* algorithm and its comparison with prior work in more detail.

- The second contribution is a *counterexample-guided invariant inference* algorithm called *Sifer*. *Sifer* is a purely static algorithm to find inductive invariants between state elements of programs that have large syntactic gap across them. It uses the concrete models (aka counterexamples) returned by the SMT solver to infer expressive invariants like affine invariants or inequality invariants with respect to a constant; such invariants were otherwise considered as intractable for static invariant inference. Another important property of the proposed *Sifer* algorithm is that it is simple enough to be used with advanced and incremental correlation algorithms like [14] or the proposed *Counter* algorithm. Unlike single-step correlation algorithms which call the invariant inference procedure once after completing the correlation for whole program, an incremental correlation algorithm invokes the invariant inference procedure multiple times at each incremental step. Thus the invariant inference algorithm used with incremental correlation algorithms in the equivalence checking setting should be efficient and should have a bounded runtime. Our proposed *Sifer* algorithm is based on Data-Flow analysis and thus is itself incremental in nature.

  Chapter 4 discusses the proposed *Sifer* algorithm, its properties like runtime bound and precision guarantees, and its comparison with prior literature in more detail.

- The third contribution is an *Unoptimized-IR-to-Optimized-Assembly* translation validation tool, *COUNTER*, based on the proposed correlation and invariant inference algorithms. To our knowledge, ours is the first black-box equivalence checking tool that can successfully compute equivalence across the unoptimized IR for a C-language program and the optimized x86 assembly program. Unlike prior work that verifies only a few selected components of the compiler or uses a pass-by-pass approach for verification, the proposed tool can verify the composition of a long and rich pipeline of transformations like loop unrolling, peeling,

unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, etc.

The proposed tool is categorized as a *black-box equivalence checker* because it makes minimal assumptions on the exact nature of transformations performed. Such an equivalence checker is capable of deciding equivalence across a larger and general space of transformations. Unlike witness generation based techniques that modify the compiler source code and put undue burden of verification on compiler developers, the black-box equivalence checker can be built independently by formal verification experts. Further, it can be used in other equivalence checking applications like program synthesis/superoptimization which require the equivalence checker to consider the transformations performed while proposing the optimized program as black-box.

A competing approach to translation validation for verifying the compiler correctness is *certified compilation* (exemplified by the CompCert compiler [42]). Certified compilation involves manually developing the compiler from scratch along with the correctness guarantees using a proof assistant. As compared to translation validation which involves verifying the input-output behavior for every compilation, certified compilation involves proving in advance that the compiler will always produce semantic preserving executable code, which requires laborious manual effort and is non-trivial for many loop optimizations. In comparison to this, a robust and comprehensive translation validation tool (like COUNTER) has significant scope for automation and can be reused to verify common off-the-shelf (COTS) compilers.

Chapter 5 discusses the implementation details for the Unoptimized-IR-to-Optimized-Assembly translation validation tool and presents its evaluation and limitations.

Chapter 2 provides the required background and the conclusion is presented in chapter 6.

Chapter 2

Preliminaries

2.1 Formal Definition of Equivalence in the context of Translation Validation

Given the pair of specification (C) and implementation(A) programs, a translation validator verifies that the observable behaviors produced by the implementation program A are equivalent to the observable behaviors produced by the specification program C. Here and in the rest of the thesis, C is used to denote the specification program and A is used to denote the implementation program. Observable behaviors include the return values of the program (e.g., exit code of the `main` procedure) and the non-temporary memory state which includes the heap and the memory region belonging to the global variables. Observable events also include any intermediate calls to procedures (aka functions) whose definition is not available in the same compilation context as its caller, because the compiler and the validator must conservatively assume that such callee procedures can potentially result in I/O events (e.g., through system calls). Without inter-procedural optimizations (our setting), all callee procedures are undefined.

In the context of translation validation, the program specification represents the language-level semantics of the program, e.g., the C language semantics of a C program, and the implementation

```
int a[100];
C0: void ubDCE(int n){
C1:   for(int i=0; i<n; i++) {
C2:     a[i] = 0;
C3:     if (i > 100)  {...}
C4:   }
C5:   return;
EC: }
```

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

models the semantics of the low-level assembly opcodes that run directly on the hardware, e.g., the semantics of the x86 assembly instructions. High-level languages (like C-language) have associated *undefined behavior (UB)* conditions. The compiler assumes the absence of undefined behaviors in the source program and uses this information to perform aggressive optimizations.

To see this with an example, consider the C program shown in fig. 2.1. In this example program, the compiler is free to remove the logic at location `C3` because accessing an array beyond its size is an undefined behavior in C-language and thus the variable i should be less than 100 for the program to be free of out of bound array access UB. The translation validator should also model the high-level language UB conditions in order to validate the transformations performed by the compiler which rely on the absence of these UB conditions.

In the presence of UB conditions in the specification program language, equivalence checking across the specification and implementation programs reduces to refinement checking: *The translation validator checks that if the specification program is safe (i.e. does not exhibit undefined behavior), then the observable behaviors produced by the implementation program are equivalent to the observable behaviors produced by the specification program.* To tackle undefined behaviors in C, a special "error" state is introduced in C which is reached whenever UB condition gets triggered.

Both the specification program C and the implementation program A are deterministic in the absence of undefined behavior; the equivalence problem in this setting can be over-approximately reduced to the checking of *weak trace equivalence*: C and A are equivalent, if for all inputs

1. *Either* both C and A programs produce identical (and potentially infinite) sequence of observable events;
2. *Or* C exhibits UB on that input and transitions to the error state which is characterized as an internal action in this formalism.

This check is over-approximate because all types of UB in C-language are not modeled. For example, the absence of non-terminating behaviors in C, which is often UB, is not modeled and C and A can potentially produce infinite sequence of observable events.

2.2 Control-Flow Graph Representation

Formal verification involves computing equivalence for all inputs, thus an abstract representation is used for the input specification and implementation programs to the equivalence checker. This section presents the details for the abstract framework named as *Control-Flow Graph* (CFG[1]) used for program representation in this thesis.

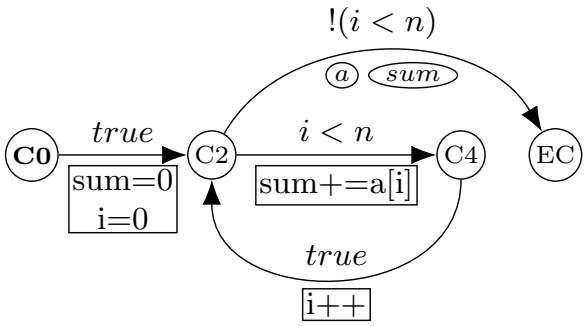
CFG is a directed graph with nodes and edges. Each node in the CFG representation of a program corresponds to a program location or program counter (PC) and is denoted by the symbol n. Each edge in the CFG corresponds to transition from one program location to another and is denoted using $\omega[n \rightarrow n']$ from node n to node n'. Each edge (representing a transition) is labeled with an *edge-condition* that must be true to trigger that transition, a *transfer function* that specifies how the (abstract) machine's state is modified across that transition, and an *action* that indicates the program's potential interactions with the environment (e.g., program exit, unspecified procedure call). The CFG for a program has a start node (n_{start}) at which it begins execution, and a (potentially empty) set of exit nodes.

The CFG representation of the program is analogous to deterministic labeled transition system. It uses a symbolic representation denoted by Ω_n to specify the state elements at node n. In this symbolic representation:

[1]Our notion of CFG is different from the standard term which is used to represent a graph over basic blocks.

```
int a[100];
C0: int array_sum(int n){
C1:   int sum = 0;
C2:   for(int i=0; i<n; i++) {
C3:     sum += a[i];
C4:   }
C5:   return sum;
EC: }
```

(a) C program



(b) C program CFG. `C0` is the entry node and `EC` is the exit node

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

- The edge-condition for edge $\omega[n \rightarrow n']$ is a boolean-valued function, $\{econd_\omega : \Omega_n \rightarrow \texttt{bool}\}$, which takes as input the symbolic state at the source node of the edge.
- The transfer function $p_\omega : \Omega_n \rightarrow \Omega_{n'}$ takes the symbolic state at source node of the edge as input and returns the symbolic state for the target node of the edge.
- The action α_ω for exit edges i.e. edges with target node as exit node, represent the return value for the procedure (if non-void) and the non-temporary memory state. For an undefined procedure call edge, α_ω includes the procedure name and its actual arguments, along with the memory-regions that the callee function may access (includes heap, global variables, and any address-taken local variables/stack). The edges which are not corresponding to a procedure call or are not exit edges, are labeled with a special internal action τ. This represents an internal behavior (or *mute transition*) that is not observable. This work does not support programs with procedure calls and thus only exit edges in the CFG are labeled with an observable action α_ω and all other edges are labeled with the internal action τ.

Figure 2.2 shows an example C-language program and its control-flow graph representation. The example program calculates the sum of the elements of the global array '`a`'. In the CFG representation shown in fig. 2.2b, the edge condition is shown above the transition edge, and the transfer function is shown inside a box below the transition edge. As an example, the edge condition for the edge `(C2 → C4)` is $(i < 100)$ and the transfer function for the edge `(C4 → C2)`

is $(i++)$. The action α for an exit edge is shown inside an ellipse below the transition edge. In this example CFG, there is a single exit edge `(C2` $\rightarrow$ `EC)` and the action α associated with this edge (shown in ellipse below the edge) returns the variable `sum` that holds the return value and the memory region for global variable `a`.

The labels `(C0)`, `(C2)`, `(EC)`, etc., in the C-program denote the program locations or PCs and are used to represent the nodes in the CFG. In this example and in the rest of the thesis, we use the following correspondence between a program's PC labels and its CFG's node names: (1) for a general statement, the PC label represents the start of the corresponding statement in the program; (2) for a `for-loop` construct, the PC label corresponds to the condition test which is also the loop head. The for-loop initialization statement is shown as a part of the transfer function of the incoming edges to the loop head; (3) `EC` and `EA` represent program exits for the specification program C and the implementation program A respectively. The CFG representation used in this thesis is similar to the *Transfer Function Graph* representation used by prior work [14].

2.2.1 Modeling Undefined Behavior

To model undefined behavior conditions, the CFG representation additionally associates each edge $\omega[n \rightarrow n']$ with a boolean valued function $\sigma_\omega : \Omega_n \rightarrow$`bool`, which takes as input the machine state at source node n. σ_ω encodes the condition that represents the absence of undefined behavior. If at node n (source node of edge $\omega[n \rightarrow n']$), the condition σ_ω is violated then the abstract machine transitions into a special `Error` node. In other words, the transition to the special `Error` node in the CFG indicates that UB was triggered. The `Error` node has no outgoing transitions and once entered, the program stays in the `Error` node forever henceforth. The edges that transition into the `Error` node are labeled with a special observable "Error" action. Also, all concrete machine states that reach the target node n' of the edge $\omega[n \rightarrow n']$ satisfy the condition σ_ω.

Figure 2.3 shows the CFG along with the UB conditions (σ_ω) for the example C program listed in fig. 2.2a. The UB conditions associated with the edges are shown in a box above the edge in boldface. In this example, the UB condition for edge (C2 $\rightarrow$ C4) is $i < 100$ because accessing an array beyond its size is undefined behavior as per C-language semantics and the size of global

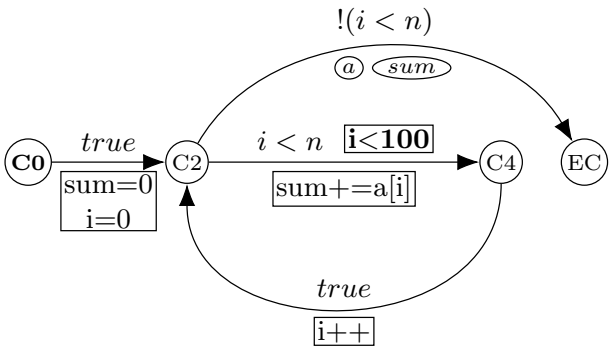


Figure 2.3: CFG with UB assumptions for C program shown in fig. 2.2a

.

array `'a'` is 100. Any value of $i \geq 100$ will trigger the undefined behavior and the abstract machine transitions to `Error` state for all such inputs. Identifying these UB conditions for a given program requires (deep) program analysis and a more detailed discussion on identifying and modeling undefined behavior is available in prior work on modeling undefined behavior[15]. In general, for completing an equivalence proof, we do not necessarily need to identify all possible UB conditions. However, the identified UB conditions should be strong enough to capture the assumptions made by the optimizing compiler.

With the UB conditions modeled in the CFG, the given pair of CFGs is considered equivalent if for each potential input:

1. Either both the CFGs are weak trace equivalent, i.e., they produce identical sequence of observable (non-internal) events.

2. Or the specification CFG transitions to the `Error` state for that input.

Also note that, if for the given input, the implementation program transitions to the `Error` state while the specification program does not, then the programs are *inequivalent* by definition. In the rest of the thesis, for simplicity and brevity, we will omit the second condition i.e. we are only interested in proving equivalence for inputs that do not trigger UB. We will henceforth assume that the equivalence needs to be proven only if the source program does not reach the `Error` state.

2.2.2 Well-formed CFG

A given control-flow graph (CFG) is *well-formed* iff the following conditions are met:

1. For every non-exit node in the CFG, the outgoing edge conditions are mutually exclusive, i.e., for all possible machine states at the node, the edge condition of exactly one of the outgoing edges at that node must be true.

   For example, in the CFG shown in fig. 2.2b, the edge conditions for the two outgoing edges of node `C2` are $i < n$ and $!\,(i < n)$, which are mutually exclusive.

2. For every non-exit node in the CFG, the union of edge-condition of all outgoing edges at that node must be *true*. Any node for which this condition is not satisfied is called an *incomplete node*.

   For example, the union of the edge conditions for the two outgoing edges of node `C2`, i.e., $((i < n) \cup !\,(i < n))$ is *true* for the CFG shown in fig. 2.2b.

3. If the set of concrete machine states that is possible at a node n is represented as a logical formula Inv_n (denoting the *node-invariant* at n), then the following relation holds for every edge $\omega[n \rightarrow n']$ in the CFG,

   $$(\sigma_\omega \wedge Inv_n \wedge econd_\omega) \Rightarrow \texttt{WP}_\omega(Inv_{n'})$$

   where $\texttt{WP}_\omega\texttt{(p)}$ represents the weakest precondition of the predicate `p` across the transition through edge ω. This well-formedness criterion ensures that the set of machine states reachable at the target node n' of edge $\omega[n \rightarrow n']$ is a superset of the set of machine states reachable at n' through n. In other words, the node-invariants in a well-formed CFG must be inductive.

2.3 Equivalence Checking Through Simulation Relation Construction

The CFG representation of the program is analogous to a deterministic labeled transition system and thus for CFGs, weak trace equivalence and weak bisimilarity coincide [59]. Most prior work on equivalence checking (including our proposed equivalence checker) is based on identifying a (weak) bisimulation relation between the given specification (C) and implementation (A) CFGs.

2.3.1 Product-CFG

The bisimulation relation, which represents a proof of equivalence across two given CFGs, can itself be represented using a control-flow graph called *product-CFG* (π). The inductive node-invariants for the product-CFG represent the inductive invariants which relate the state-elements of the two programs. Figure 2.4a shows an example C-language program with nested loops. The program computes the sum of a two-dimensional array 'a' and returns the computed sum. Figure 2.4b shows an equivalent abstracted assembly language program. The assembly program has undergone many optimizations including loop inversion for both the loops, loop splitting, and register allocation. Figure 2.4c and Figure 2.4d show the CFG representation for the C-program and the assembly program respectively. One of the possible product-CFGs across the two programs along with the inductive invariants is shown in fig. 2.4e and fig. 2.4f respectively.

The product-CFG encodes the lockstep execution of the two given programs represented as (deterministic) CFGs and has the following properties:

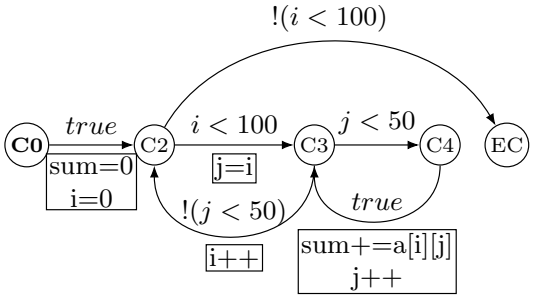
1. A node in the product-CFG is formed by pairing two respective nodes (or PCs) of C and A CFGs and is termed as a *PCpair*. For instance, the node `(C2,A2)` in the product-CFG in fig. 2.4e is formed by pairing node `(C2)` of C-program CFG and node `(A2)` of assembly program CFG. A PCpair indicates that if an abstract machine visits a node (n_C, n_A) in product-CFG, then it would have visited n_C in C and n_A in A.
2. The machine state for the product-CFG is formed by pairing the machine states of C and

```
int a[100][50];
C0: void nestedLoop(){
C1:   int sum = 0;
C2:   for(int i=0; i<100; i++) {
C3:     for(int j=i; j<50; j++) {
C4:         sum += a[i][j];
C5:     }
C6:   }
C7:   return sum;
EC: }
```

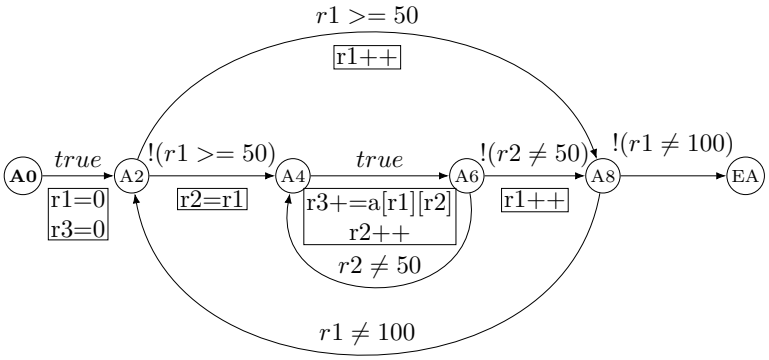
(a) C program

```
A0: nestedLoop:
A1:  r1 = 0; r3 = 0
A2:    if (r1 >= 50) goto A7
A3:    r2 = r1
A4:      r3 += a[r1][r2]
A5:      r2++
A6:      if (r2 != 50) goto A4
A7:    r1++
A8:    if (r1 != 100) goto A2
EA:  ret r3
```

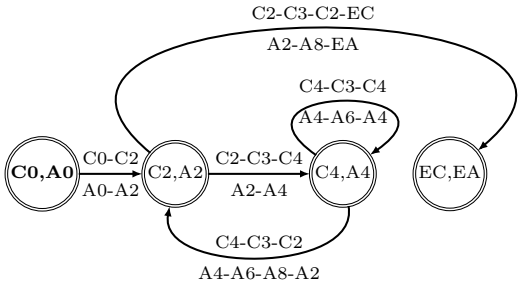
(b) (Abstracted) Assembly program



(c) C program CFG.



(d) Assembly program CFG.



(e) One of the possible Product-CFG

Node	Invariant
(C0,A0)	$\mathtt{H_C} = \mathtt{H_A}$
(C2,A2)	$\mathtt{sum} = \mathtt{r3}; \mathtt{i} = \mathtt{r1}; \mathtt{H_C} = \mathtt{H_A}$
(C4,A4)	$\mathtt{sum} = \mathtt{r3}; \mathtt{i} = \mathtt{r1}; \mathtt{j} = \mathtt{r2}; \mathtt{H_C} = \mathtt{H_A}$
(EC,EA)	$\mathtt{sum} = \mathtt{r3}; \mathtt{H_C} = \mathtt{H_A}$

(f) Inductive invariants for product-CFG in fig. 2.4e

Figure 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.

A CFGs. For example, product-CFG's machine state has two memories corresponding to the memory state of C and A programs.

3. An edge in the product-CFG is formed by pairing a path (or a series of edges) in the two CFGs. As an example, the product-CFG edge $(\texttt{C0}, \texttt{A0}) \rightarrow (\texttt{C2}, \texttt{A2})$ is formed by pairing the path $(\texttt{C0} \rightarrow \texttt{C1} \rightarrow \texttt{C2})$ in C with the path $(\texttt{A0} \rightarrow \texttt{A1} \rightarrow \texttt{A2})$ in A.

   A product-CFG edge encodes the lock-step execution, i.e., the product-CFG edge $\omega = (n_C, n_A) \rightarrow (n'_C, n'_A)$*, represents that* C *makes a transition from* $n_C \rightarrow n'_C$ *and* A *makes a transition from* $n_A \rightarrow n'_A$.

4. The edge condition of an edge in the product-CFG is formed by conjuncting the path conditions for the corresponding paths in C and A. The path condition is computed as the weakest-precondition of the `True` predicate on the given path. The weakest-precondition (`WP`) for any predicate P on a series composition ($\odot$) of two edges ω_1 and ω_2 can be computed as follows:

$$\texttt{WP}_{\omega_1 \odot \omega_2}(P) = \texttt{WP}_{\omega_1}(\texttt{WP}_{\omega_2}(P))$$

5. The transfer function of an edge in the product-CFG is formed by composing the transfer function for the corresponding paths in C and A. The transfer function for a path is computed by composing the transfer function of individual edges as follows:

$$p_{\omega_1 \odot \omega_2}(\Omega) = p_{\omega_1}(p_{\omega_2}(\Omega))$$

6. The node-invariant at the start node of the product-CFG is formed by equating the program arguments and non-temporary memory states of C and A programs while accounting for the different input representations in the two syntaxes. The first row in fig. 2.4f shows the node-invariant at the start node `(C0,A0)` of the product-CFG, where `H` represents the non-temporary memory state for a program which includes the memory regions belonging to the global variables and heap.

7. The product-CFG is a well-formed CFG and satisfies the well-formedness conditions outlined in section 2.2.2.

The second well-formedness condition states that all non-exit nodes should be complete i.e. the union of edge conditions for all outgoing edges from a node in the CFG should be true. *This well-formedness condition in the context of product-CFG implies that a well-formed product-CFG includes all possible behaviors of the (deterministic) CFG pair on legal inputs and hence serves as a proof (or witness) of bisimulation between C and A CFGs for all legal inputs to C and is also termed as* **complete product-CFG**.

The third well-formed condition states that the node-invariants in a well-formed CFG are inductive. Thus, given the node-invariant at the start node of the product-CFG, the node-invariants at other nodes of the product-CFG can be identified through a sound and over-approximate inference procedure. *If the inferred invariants are strong enough to prove equivalent observable behavior, then the product-CFG along with the inferred invariants represent the* **witness (or proof)** *of equivalence.* For example, the second and third row in fig. 2.4f show the inferred inductive invariants at the product-CFG nodes `(C2,A2)` and `(C4,A4)` respectively and are strong enough to inductively prove the equivalence of observables (the return value and the non-temporary memory state in this case) represented using the invariant at exit-node `(EC,EA)`.

Most prior work on simulation relation construction based equivalence checking involves constructing the complete product-CFG in a single step mostly using a common alignment condition [63, 11]. An alignment condition or alignment predicate (AP) can be thought of as a relation between the state elements of the specification program C and implementation program A such that for all edges $e = (\eta_C, \eta_A)$ in the product-CFG, the states at the two (start and stop) endpoints of the paths η_C in C and η_A in A are related by AP. The single step construction of the complete product-CFG simplifies the simulation relation construction problem by partitioning it into two completely disjoint steps for correlation and invariant inference. The algorithm first constructs the complete product-CFG by identifying the correlated transitions that cover all possible program behaviors followed by an invariant inference algorithm to infer the inductive node-invariants for each node in the static product-CFG. The advantage of performing invariant inference on the complete product-CFG is that sophisticated invariant inference techniques (for example using constraint solving [13, 26]) can now be used for relating the state-elements at each node of the product-CFG. However, the biggest shortcoming of this approach is that different fragments (or loops) in a program may require different alignment conditions for correlation and this approach

cannot handle such programs by correlating the whole program in a single step using a common alignment condition.

2.3.2 Incremental Algorithm for Product-CFG Construction

To overcome the shortcomings of simulation relation construction algorithms which involve constructing the complete product-CFG in a single step mostly using a common alignment condition, an incremental algorithm for product-CFG construction was proposed by Dahiya et al. [14]. The pseudo code for the incremental algorithm proposed by Dahiya et al. is shown in fig. 2.5. The algorithm takes as input the CFG for the specification program (C) and the CFG for the implementation program (A). It returns either a product-CFG that is a provable bisimulation (if proof found) or null (if the proof was not found).

The top-level procedure **incrementalCorrelate()** initializes a partial product-CFG π_{init} to a CFG that has a single node `(C0,A0)` formed by pairing the start node `C0` in C and the start node `A0` in A. The procedure then "expands" the partial product-CFG π_{init} by adding new product-CFG edges to it through the call to `expandProductCFG()`. As noted in section 2.3.1, each of these product-CFG edges is formed by pairing a path from the two CFGs. The `expandProductCFG()` procedure enumerates multiple potential path pairs for each product-CFG edge to be added and uses a backtracking search tree to store these possible correlations. The *frontier* Ω of the backtracking search tree is initialized by a call to `expandProductCFG()` procedure for the initial product-CFG (π_{init}).

The **expandProductCFG()** procedure takes a partial product-CFG π as input and through a call to `findIncompleteNode()` procedure identifies an incomplete product-CFG node $n = (n_C, n_A)$ in π. As discussed in section 2.2.2, an incomplete node in the CFG is a node for which the union of edge-condition of all outgoing edges at that node is not true. If no such node exists in the input partial product-CFG π then the `findIncompleteNode()` procedure returns a `nullptr` indicating that the input product-CFG π is already complete; in which case, the `expandProductCFG()` procedure returns to the caller which will eventually check if the inferred invariants ensure equivalent observable behavior (`ProductCFGisProvableBisim()`).

Algorithm 1: Incremental Product-CFG Construction Algorithm

```
1  Function expandProductCFG(π, C, A, Ω)
2      if ¬ ((n_C, n_A) ← findIncompleteNode(π)) then
3          return True;
4      end
5      path_A ← getNextPath(n_A, A);
6      κ_C ← getCandCorrelations(n_C, C, path_A);
7      foreach path_C ∈ κ_C do
8          if actionsAreCompatible(path_C, path_A) then
9              π_new ← π;
10             addEdge(π_new, (path_C, path_A));
11             Ω ← Ω ∪ {π_new};
12         end
13     end
14     return False;
15 Function incrementalCorrelate(C, A)
16     π_init ← initProductCFG(C, A);
17     Ω ← { };
18     expandProductCFG(π_init, C, A, Ω);
19     while Ω is not empty do
20         π_cur ← removeDepthFirstOrder(Ω);
21         if checkCriterionForNewEdge(π_cur) then
22             inferInvariants(π_cur);
23             if checkCriterionForEdges(π_cur) then
24                 isComplete ← expandProductCFG(π_cur, C, A, Ω);
25                 if isComplete ∧ ProductCFGisProvableBisim(π_cur) then
26                     return π_cur;
27                 end
28             end
29         end
30     end
31     return null;
```

Figure 2.5: Pseudo code for the incremental product-CFG construction algorithm

If there exists an "incomplete node" $n = (n_C, n_A)$ in the partial product-CFG π, the **getNextPath()** procedure is used to identify the path $\texttt{path}_A$ starting at node n_A in A that has not yet been correlated. The **getCandCorrelations()** procedure then enumerates multiple potential correlations for $\texttt{path}_A$ in the specification C starting at node n_C and returns these multiple possible correlation candidates in κ_C. A product-CFG edge is formed by pairing the chosen path $\texttt{path}_A$ in A with one of the possible correlation candidate $\texttt{path}_C \in \kappa_C$. The detailed algorithm to enumerate the set of paths to be correlated in A and the multiple possible correlation candidates in C is given in section 4.3 in [14] and is briefly discussed here.

The enumeration algorithm for A involves fixing the entry, exit and loop heads in A as anchor PCs (or nodes) and the path between these anchor nodes form the set of paths $\{\texttt{path}_A\}$ to be correlated in A. For example, the chosen anchor PCs for the implementation CFG shown in fig. 2.4d are `(A0, A2, A4, EA)` and the set of paths that need to be correlated is {`(A0-A2)`, `(A2-A4)`, `(A4-A4)`, `(A4-A2)`, `(A2-EA)`}.

The enumerated correlation candidates κ_C in the specification program C for a chosen $\texttt{path}_A$ in A consist of composite paths from the node n_C to any other node in C up to a bounded unroll factor. A bound on the unroll factor ensures that an enumerated path must be smaller than a multiple of the program size; in practice, the enumerated paths are much shorter due to the presence of intermediate anchor nodes.

A composite path between two nodes is formed by composing a sequence of edges (into a path), or by combining a disjunction of multiple paths. The disjunction of multiple paths is required to handle transformations which involve merging of multiple paths in the source program to a single path in assembly program. These transformations are mainly possible because of the availability of conditional opcodes in the assembly syntax; the code generated using these opcodes is more performant and it enables more optimization opportunities. An example program-pair for which the required product-CFG consists of composite path involving disjunction of multiple paths in the specification program is shown in fig. 3.1. The required composite path in this example consists of 81 individual paths and we discuss this example in detail in section 3.2.1.

In general, the total number of paths that are required to be correlated in a single composite

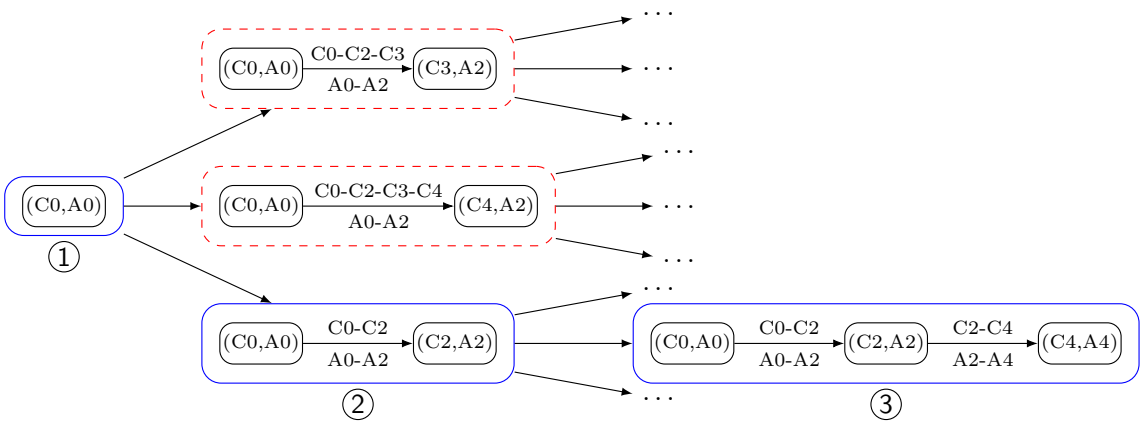


Figure 2.6: A snapshot of the backtracking search tree for the program-pair shown in fig. 2.4

path can be exponential in the size of the program and unrolling performed by the compiler. We present techniques to scalably handle this exponential number of paths requirement in our proposed *counterexample-guided correlation algorithm*.

For each enumerated correlation candidate $\texttt{path}_C \in \kappa_C$, first the compatibility of its action is checked with the action on $\texttt{path}_A$ (`actionsAreCompatible()`). For programs without function calls (as considered in this thesis), the action compatibility check involves checking that $\texttt{path}_C$ is an exit path iff $\texttt{path}_A$ is an exit path (as an exit path can potentially produce an "observable action"). If the action compatibility check passes, a new product-CFG π_{new} is created which additionally includes the new product-CFG edge encoding the candidate correlation between $\texttt{path}_C$ and $\texttt{path}_A$.

All these newly created product-CFGs (in the `expandProductCFG()` procedure) containing one additional edge as compared to the input product-CFG π are added back to the frontier Ω of the backtracking search tree[2]. A snapshot of the search tree for the pair of programs in fig. 2.4 is depicted in fig. 2.6. Each node in the search tree represents a partially-constructed product-CFG, and the outgoing edges at a node represent the potential possibilities for the newly added edge. Here, the frontier Ω represents the set of all partially-constructed product-CFGs at the end of all visited paths.

[2]The frontier Ω is passed as a *call by reference* parameter to the `expandProductCFG()` procedure.

Even if finite number of candidate correlation possibilities are added to the frontier at each step (i.e. each call to `expandProductCFG()` function), this incremental approach for simulation relation construction results in significantly large (potentially exponential) number of possible product-CFGs. In order to build a robust correlation procedure, an exhaustive search based approach is used to find the required product-CFG in this exponentially-large space; it involves iteratively choosing each of the enumerated partial product-CFGs until the required product-CFG that yields a provable bisimulation relation is found or the complete exponential search space is exhausted. Dahiya et al. [14] used a depth-first order (`removeDepthFirstOrder()`) to iteratively pick a product-CFG from the frontier. This approach exhaustively explores a subtree in the backtracking search tree before attempting another subtree of possible correlations. In section 5.2, we empirically show that the depth-first search based correlation becomes intractable for aggressive optimizations that result in large structural differences between the program-pair.

We present techniques to efficiently find the required product-CFG from the exponential search space in our proposed *counterexample-guided correlation algorithm*. The algorithm proposes a counterexample-based pruning and ranking strategy that can efficiently search (*best-first search*) this space to identify a provable bisimulation relation. The details of these counterexample-based pruning and ranking algorithms are presented in section 3.3. We also make empirical comparison of the depth-first order used by the prior work and the proposed best-first search procedure in section 5.2.

An alignment condition (also referred as correlation criterion) is checked for the newly added edge (`checkCriterionForNewEdge()`) in the chosen partial product-CFG π_{cur}. Since the correlation criterion is checked for each new edge being added as part of the incremental construction algorithm, more sophisticated heuristics which may depend on the specific edge being correlated can be used. This lends robustness to the algorithm and simulation relation for more complex program-transformation pairs can be established using the incremental approach. If the newly added edge satisfies the alignment condition, then the algorithm updates the node-invariant at the target node of the newly added edge through a call to **inferInvariants()** procedure. Note that, the invariant inferred at a product-CFG node refers to the strongest inductive relation between variables (state-elements) of the program-pair at that node in the chosen grammar and is not guaranteed to be strong enough to prove equivalence for the complete product-CFG. If

the chosen correlations and the inferred inductive invariants ensure equivalent observable behavior, then we obtain a proof (or witness) of equivalence; otherwise the algorithm backtracks and tries to prove equivalence using the other possible correlation candidates. We have chosen our invariant-inference grammar carefully so that the invariants drawn from it are usually sufficient to complete equivalence proofs across typical compiler transformations.

The node-invariants associated with product-CFG nodes other than the target node of the newly added edge were inferred during the previous call to the `inferInvariants()` function. These invariants may not be inductively provable now after the new edge has been added. Thus, the `inferInvariants()` procedure also re-runs the inference and updates the invariants at these nodes as well. Since the invariant inference algorithm is invoked multiple times during the product-CFG construction, sophisticated invariant inference algorithms which do not have termination guarantees or which rely on a static product-CFG and are very expensive if re-run on the updated product-CFG, cannot be used with incremental simulation relation construction approach. At each incremental step, the invariant inference algorithm is re-run for all nodes in the partial product-CFG to update the relations after adding a new edge.

Our proposed *counterexample-guided invariant inference* algorithm is based on a data-flow analysis framework and performs incremental computation (instead of starting from scratch) when re-run for updating the already inferred invariants after adding the new edge. However, the proposed algorithm is still powerful enough to infer expressive invariants (like affine invariants) using a purely static approach. The details of our proposed invariant inference algorithm is presented in section 4.2.

After invariant inference, the alignment condition (or the respective correlation criterion) for all edges is checked again using `checkCriterionForEdges()` function, as the alignment conditions may not hold after updating the invariants. If the alignment condition of any of the edge is not satisfied after invariant inference then the algorithm *backtracks* and attempts the next potential correlation candidate from the frontier. If the alignment condition still holds for all edges in the partial product-CFG and all the paths are already correlated in the partial product-CFG constructed so far (i.e. the product-CFG is complete) and the inferred invariants are strong enough to prove equal observables across the two programs, then the product-CFG (including

the inductive node-invariants) is returned by the algorithm and represents the proof (or witness) of equivalence.

2.4 Counterexample

As discussed in section 2.3, the equivalence checking through product-CFG construction involves two major steps:

1. The first step is to correlate transitions across the given program-pair such that the states at the end-points of these transitions remain related. The correlated transitions are added to the product-CFG as an edge and the end-points of these correlated transitions form the product-CFG nodes.
2. The second step is to identify the relations between the state-elements of the program-pair at the product-CFG nodes. These relations are referred as node-invariants.

The node-invariant at the start node of the product-CFG is formed by equating the program arguments and non-temporary memory states of C and A programs. For example, the first row in fig. 2.4f shows the node-invariant at the start node `(C0,A0)` of the product-CFG, where `H` represents the non-temporary memory state. Also, as noted earlier, the node-invariants in a well-formed product-CFG are inductive and thus given the node-invariant at the start node of the product-CFG, the node-invariants at other nodes of the product-CFG are usually identified through a sound and over-approximate inference procedure. This invariant inference procedure involves guessing or inferring a possible invariant (or a relation between the state-elements) at a product-CFG node and checking if the inferred invariant is inductively provable across all incoming edges of that product-CFG node.

Given an inferred invariant $\texttt{Inv}_{n'}$ at the product-CFG node n' and an incoming edge $\omega = n \rightarrow n'$, the check for the inductive property is represented as a relational Hoare triple $\{\texttt{Inv}_n\}\omega\{\texttt{Inv}_{n'}\}$ at node n. Here, $\texttt{Inv}_n$ represents the invariant at node n. This Hoare triple states that if the machine starts at node n such that it satisfies $\{\texttt{Inv}_n\}$, and the edge ω is executed, then the resulting machine state would satisfy $\{\texttt{Inv}_{n'}\}$. To discharge proof obligations, the Hoare triple

$\{\texttt{Inv}_n\}\omega\{\texttt{Inv}_{n'}\}$ is converted (or lowered) to a propositional boolean logic formula at node n of the form $\texttt{Inv}_n \Rightarrow \texttt{WP}_\omega(\texttt{Inv}_{n'})$, where $\texttt{WP}_\omega(\texttt{Inv}_{n'})$ computes the weakest precondition of $\texttt{Inv}_{n'}$ across ω. The proof obligation for this Hoare triple boolean formula is discharged through an off-the-shelf SMT solver with quantifier-free bitvector, array and uninterpreted function theories. If the proof succeeds, $\texttt{Inv}_{n'}$ is an inductively-provable invariant across the edge $\omega = n \rightarrow n'$.

If the proof obligation does not succeed, then a counterexample γ_n at node n is returned by the SMT solver. The generated counterexample γ_n has a concrete assignment to the program variables or state-elements at the node n such that it satisfies the invariant at the node where it is generated (i.e. $\texttt{Inv}_n$) and it represents a concrete machine state that may occur at that node during real execution. In the context of a product CFG, the generated counterexample would involve concrete valuations of variables for both the specification C and the implementation A programs.

The generated counterexample is then propagated on the edge ω for which inductive query was made by applying the concrete execution function p_ω for this edge. Since, the generated counterexample γ_n does not satisfy the weakest-precondition of the candidate invariant across the edge ω, i.e., $\texttt{WP}_\omega(\texttt{Inv}_{n'})$, the propagated concrete machine state does not satisfy the candidate invariant $\texttt{Inv}_{n'}$ and can be used to refine the guessed (or inferred) invariant. We will see details for this later in section 4.2.

The propagation of a counterexample is quite similar to the interpreted execution of the product-CFG on a concrete machine state, with two operational differences:

1. Counterexamples need not have concrete valuations for all live program variables — they just contain valuations for those variables that were a part of the SMT query that generated it. Thus, during propagation, if the program reads a variable that is not already present in the counterexample, a random value is generated for that variable and added to the counterexample.

2. The potential presence of UB in the specification program may interfere with counterexample propagation. If a counterexample triggers UB during propagation, we do not propagate it any further — in other words, the counterexample transitions to the special “error” state

meant to catch UB. Thus, a counterexample differs from inputs derived from real program traces: while real traces must never trigger UB, a counterexample generated through an SMT query has no such requirement.

The main insight of this thesis is that – *Even though counterexamples cannot replace real execution traces, a counterexample is still useful because it satisfies the inferred invariant at the node at which it was generated, and it does not trigger UB on the paths on which it is propagated.* Thus it is safe to consider a counterexample at par with a real concrete machine state for these smaller program segments where it does not trigger UB, because our reasoning power in these smaller segments is constrained by the inferred invariants in any case.

Based on this insight, we propose a *Counterexample-guided correlation algorithm* that can efficiently find the correlation between program transitions for structurally significantly different (but bisimilar) program-pairs and a *Counterexample-guide invariant inference* algorithm to find a general class of relations between state elements of programs that have significant syntactic gap across them. Using both these algorithms, we demonstrate 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. Our proposed black-box equivalence checker can statically compute equivalence in less than 1% time for benchmarks used by prior work [11] and is able to handle many more program-transformation pairs.

Chapter 3

Counterexample-Guided Correlation Algorithm

3.1 Introduction

The problem of identifying the correlated program transitions across the specification program CFG C and the implementation programs CFG A is NP-hard and is very challenging in the presence of complex transformations like loop splitting, loop unrolling, and loop unswitching because these transformations result in significant structural differences between the two programs. The space of possible product-CFGs to be considered to correlate these structurally apart program-pairs is huge. This search space can be reduced using the following three observations:

- **Observation-A**: For most programs/compiler-transformations, the maximum length of a path that needs to be correlated within a single product-CFG edge is bounded . For example, compilers bound the *unroll factor* while transforming programs. To bound correlated path lengths, we introduce a parameter, μ_C, which represents the maximum number of times a PC may appear in a program path of C that is correlated through a product-CFG edge. We only count a PC to "appear" in a program path if it is present as the target of an

edge in that path. For example, the path `(C4-C3-C4)` can be correlated at $\mu_C = 1$ because both `C3` and `C4` appear only once in it. On the other hand, the path `(C4-C3-C4-C3-C4)` cannot be correlated at $\mu_C = 1$ because both `C3 and C4` appears twice in it. Section 3.3.4 explains the parameter μ in more detail.

- **Observation-B**: It usually suffices to restrict the correlated PCs (that constitute the PC-pairs) to the heads (first instruction) and tails (last instruction) of the basic blocks of specification program's CFG and to the loop heads of the implementation program's CFG. Intuitively, even if the "ideal" product-CFG (that yields a provable bisimulation) required a PC n in the middle of a basic block to be correlated, in most cases a product-CFG that instead correlates either the head or the tail of the corresponding basic block (that contains n) also yields a provable bisimulation. A powerful invariant inference procedure that can efficiently generate expressive invariants can bridge this gap between the ideal correlation and a correlation that only considers loop heads in one program and basic block heads and tails in the other.
- **Observation-C**: For a given PCpair, it is rare for an outgoing path in A to be correlated with multiple paths in C such that these (multiple) correlated paths in C have different endpoints, but not vice-versa. Intuitively, this is so because optimizers may specialize program paths in the specification program to yield two or more versions of the same path in the optimized implementation (e.g., loop splitting, peeling, unrolling, and unswitching). Conversely, it is relatively rare for an optimizer to combine two different paths in unoptimized program C into a single path in optimized program A. This latter category of "de-specializing" transformations are usually only relevant while optimizing for code size, and are relatively rare.

In a nutshell, the above observations (A-C) reduce the number of possible correlations by (a) using an upper bound for the maximum length of path used for correlation in a single product-CFG edge; (b) restricting the correlated nodes in the product-CFG to be formed by pairing the nodes drawn from a chosen set of PCs for each input CFG; and (c) adding a single product-CFG edge for a given path in A. Although these three observations significantly reduce the number of possible correlations at each step of an incremental correlation algorithm, the whole search space still remains exponential in the size of the program and unroll factor. For example, the number of

potential product-CFGs are in the order of 10^{14} for $\mu_C = 8$ for the pair of programs in fig. 2.4. If we optimistically assume that for a given product-CFG, it takes an average of only one second to infer the invariants (and check equivalence of observables), a naive exhaustive search algorithm (as shown in fig. 2.5) can take close to 10^6 years to compute equivalence for this small example.

In this section, we present a counterexample-guided best-first search algorithm, called ***Counter*** *to efficiently search the space of potential product-CFGs to yield a provable bisimulation relation.* The name *Counter* is intended to represent two facts about the algorithm: (1) it uses counterexamples to identify the most promising correlation, and (2) it counts the number of related variables across the two programs to rank the potential correlations in the order of their promise. In section 5.2, we empirically show that the exhaustive search based correlation becomes intractable for aggressive optimizations that result in significant structural differences between the program-pair and compare our proposed *best-first search procedure* with the depth-first order used by the prior work [14].

3.2 Related Work and Motivating Examples

The identification of the product-CFG (or correlation) for the construction of a bisimulation proof has been investigated by multiple researchers previously.

Early efforts used simple branch correlation heuristics [52, 77] to construct the product-CFG. Subsequently, data-driven heuristics were proposed [63] to identify a one-to-one correspondence between the loop heads in C and A. All these approaches identify a relation between program PCs instead of program transitions and hence are inadequate for transformations involving loop peeling and unrolling.

Dahiya et al. [14] proposed the incremental simulation construction algorithm discussed in section 2.3.2. It involves a step-by-step construction of the required product-CFG and at each incremental step a new edge is added and invariants are inferred for the partially-constructed product-CFG. They used an exhaustive search based approach to identify the required transition pair among the various correlation possibilities. A correlation criterion is used by the algorithm

to restrict these correlation possibilities at each step: a new edge (representing a correlation of paths in C and A) is added to the incrementally-constructed product-CFG only if the *path conditions* (the weakest conditions under which the paths are taken) are provably identical from the invariants inferred so far. The proposed algorithm due to its incremental approach is robust and can be used for programs with multiple loops. But we identify two shortcomings of this algorithm:

1. The first shortcoming is that the number of paths that need to be correlated for an edge in the product-CFG can be exponential in the size of the program and the loop unroll count.

2. The second shortcoming is the identical path conditions requirement for correlation; we find this condition too restrictive and cannot accommodate transformations like loop splitting or all transformations that involve code specialization like loop unswitching.

This second shortcoming of the correlation algorithm proposed by Dahiya et al. [14] was addressed by Churchill et al. [11] through a data-driven algorithm named *semantic program alignment* (SPA). SPA algorithm starts by first "guessing" an *alignment predicate* (AP) that must hold at all nodes of the required product-CFG. It uses the concrete execution traces such that, if for a given input, the program C takes path η_C and program A takes path η_A such that the alignment predicate (AP) is satisfied by the machine states of C and A at the endpoints of η_C and η_A respectively, then a transition (product-CFG edge) that correlates the two paths, η_C and η_A, is added to the *Program Alignment Automaton* (PAA). In other words, an edge $e = (\eta_C, \eta_A)$ is added to the PAA only if the states at the two (start and stop) endpoints are related by AP and the programs C and A are known to take these paths η_C and η_A respectively for the same input (for all input concrete execution traces). In a nutshell, the key idea is to extrapolate the behavior of the two programs on a small set of concrete traces by using a "good" AP guess, to all possible executions on C and A. For all possible PAAs that can be constructed using the chosen AP, inductive invariants are inferred on the PAA (which is identical to a product-CFG) and the equivalence proof is completed if the inferred invariants guarantee observable equivalence for any one of these possible PAAs.

This approach is best-effort because it requires execution traces with adequate path coverage on both C and A; we find that adequate coverage may require traces that exhibit an exponential

number of distinct behaviors. Further, it relies on a good AP guess: an AP that is too strong would ignore the required product-CFG while an AP that is too weak would result in too many satisfying product-CFGs, of which most would be incapable of yielding a provable bisimulation. The authors synthesized potential APs through a syntactic grammar and evaluated their technique on mostly single-loop source programs with no control flow within the loop bodies of the source program. Churchill et al. acknowledge in their paper [11] that they leave the problem of synthesizing the required AP for programs with multiple loops (in the source program) for future work.

We motivate our algorithm by using examples that provide more details for the above mentioned shortcomings of state-of-the-art correlation algorithms, especially the correlation algorithm proposed by Dahiya et al. [14] and the SPA algorithm [11], which we think are our closest competing algorithms in terms of its capabilities.

3.2.1 Motivating Example 1

The first shortcoming of the state-of-the-art correlation algorithms is that these algorithms identify correlations for each program path individually, which is not scalable because the number of potential program paths can be exponential in program size and unroll factor. Further, SPA algorithm [11] correlates a path in the PAA only if it is seen to be taken in one of the concrete execution traces. Thus, for exponential number of paths, SPA algorithm would require traces with an exponential number of distinct behaviors to arrive at the required product-CFG (or PAA).

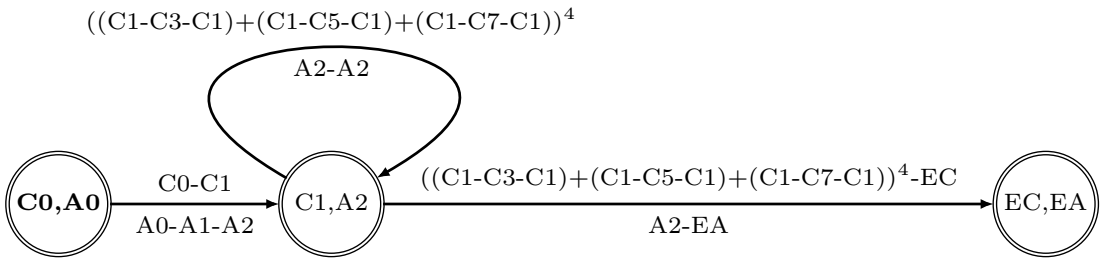
Consider the example program from TSVC suite [45] listed in fig. 3.1. The loop body in the C program shown in fig. 3.1a is unrolled four times and vectorized in the assembly code in fig. 3.1b. The edge in the product-CFG is required to correlate this loop edge in the assembly program with four unrolled loop body in the specification (or source) program. Due to 3-way control flow in the loop body in the source program, the four unrolled loop consists of $3^4 = 81$ distinct paths. For ease of exposition, we show an assembly program with four unrolling in fig. 3.1, even though the actual unrolling that can be performed by a compiler can be eight or even higher, in which case the number of paths in C that need to be correlated can be $3^8 = 6561$ (for 8 unrolling) or higher. Thus, state-of-the-art correlation algorithms would require to correlate a single edge in

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:  for (int i=0; i<LEN; i++){
C2:   if (d[i] < 0) {
C3:     a[i] += b[i] * c[i];
C4:   } else if (d[i] == 0) {
C5:     a[i] += b[i] * b[i];
C6:   } else {
C7:     a[i] += c[i] * c[i];
C8:   }
C9:  }
EC:}
```

(a) C program.
`LEN` is a positive multiple of 4.

```
A0: s441:
A1:  r1 = 0
A2:   xmm1 = a[r1 .. r1+3]
A3:   xmm2 = xmm1 + b[r1 .. r1+3]*c[r1 .. r1+3]
A4:   xmm3 = xmm1 + b[r1 .. r1+3]*b[r1 .. r1+3]
A5:   xmm4 = xmm1 + c[r1 .. r1+3]*c[r1 .. r1+3]
      // pcmpgtd
A6:   xmm0 = (d[r1] < 0), .. , (d[r1+3] < 0)
A7:   xmm1 = xmm0 ? xmm2 : xmm1  // pblendvb
      // pcmpeqd
A8:   xmm0 = (d[r1] == 0), .. , (d[r1+3] == 0)
A9:   xmm1 = xmm0 ? xmm3 : xmm1  // pblendvb
      // pcmpgtd
A10:  xmm0 = (d[r1] > 0), .. , (d[r1+3] > 0)
A11:  xmm1 = xmm0 ? xmm4 : xmm1  // pblendvb
A12:  a[r1 .. r1+3] = xmm1
A13:  r1 += 4
A14:  if (r1 != LEN) goto A2
EA:  ret
```

(b) (Abstracted) Assembly as generated by GCC



(c) Product-CFG

Figure 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.

the assembly program `(A2-A2)` (edge from `A2` to `A14` and back to `A2`) with exponential number of distinct paths $3^4 = 81$ (or $3^8 = 6561$ in case of 8 unrolling) in the specialization program C, which is inefficient and expensive. Further, being a data-driven algorithm, SPA would require traces with at least 6561 distinct behaviors (where each behavior corresponds to the traversal

of a different path in C) to be able to propose the required PAA. We find that none of the 28 benchmarks used to evaluate SPA involved control flow inside the loop bodies of the source program, and so this exponential-path problem could not get exposed during SPA's evaluation.

To address these limitations of the prior work, our proposed correlation algorithm, *Counter*, attempts to correlate a pathset in C with a pathset in A, where a single pathset may potentially represent a large number of program paths. It uses a *series-parallel digraph* representation for pathsets to correlate them efficiently across C and A in a single step. In this representation, the `+` operator indicates parallel composition, and the serial composition is denoted by simply concatenating the edges consecutively using `-` operator in a string. ϵ represents the empty path and a numeric superscript P^n is used to indicate P serially composed with itself n times. For example, `(C1-(C2-C3)`^{2}`-(`ϵ`+C4)-C5)` represents a set of two paths, namely `(C1-C2-C3-C2-C3-C5)` and `(C1-C2-C3-C2-C3-C4-C5)`. We present a detailed discussion on pathset correlation in section 3.3.3. The product-CFG generated using the proposed pathset correlation based *Counter* algorithm for the pair of programs in figs. 3.1a and 3.1b is shown in fig. 3.1c.

3.2.2 Motivating Example 2

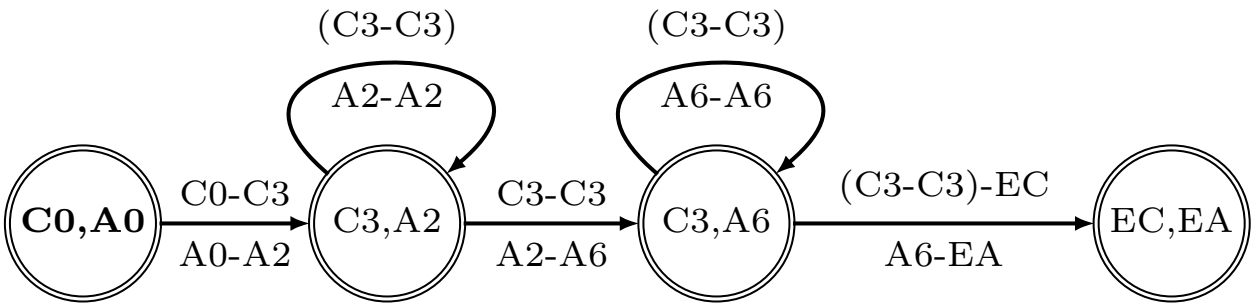
The second motivating example discusses in more detail the limitation of the identical path-conditions requirement of the correlation algorithm proposed by Dahiya et al. [14]. We find this correlation condition too restrictive as it cannot accommodate transformations like loop splitting or all transformations that involve code specialization like loop unswitching. Figure 3.2 shows an example program-pair which undergoes loop splitting and unswitching transformation and the required product-CFG across the program-pair. The product-CFG shown here correlates a pathset in the assembly program with a pathset in the C program and the conditions for the correlated loop pathsets are not equal in this product-CFG. For instance, the pathset condition for the loop `(A2-A2)` in assembly program is (`r1` $<$ `LEN/2`), whereas the pathset condition for the correlated loop `(C3-C3)` in the unoptimized C program is (`i` $<$ `LEN`) (where the inductive node-invariant at PCpair `(C3,A2)` relates (`i` $=$ `r1`)). Since the required product-CFG (which yields a provable bisimulation) violates the equal path(set) condition requirement, it will be rejected

```
int LEN, a[LEN], b[LEN];
C0: int loopSplitting() {
C1:   int sum = 0;
C2:   int mid = LEN/2;
C3:   for (int i = 0; i < LEN; i++) {
C4:     if (i < mid) sum += a[i];
C5:     if (i >= mid) sum += b[i];
C6:   }
C7:   return sum;
EC: }
```

(a) C program.
LEN is a positive multiple of 4.

```
A0: loopSplitting:
A1:   r1 = 0; r2 = 0;
A2:     r2 += a[r1]
A3:     r1++
A4:     if (r1 != mid) goto A2
A5:   r3 = 0
A6:     r3 += b[r1]
A7:     r1++
A8:     if (r1 != LEN) goto A6
A9:   r0 = r2
A10:  r0 += r3
EA:   ret r0
```

(b) (Abstracted) Assembly code



(c) Product-CFG

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

by the correlation algorithm proposed by Dahiya et al. and the algorithm will not be able to establish equivalence in this case.

The proposed *Counter* algorithm uses a more general correlation criterion which can handle a much larger class of transformations including loop splitting (bisimilar cases) and loop unswitching. The correlation criteria is discussed in more detail in section 3.3.5.

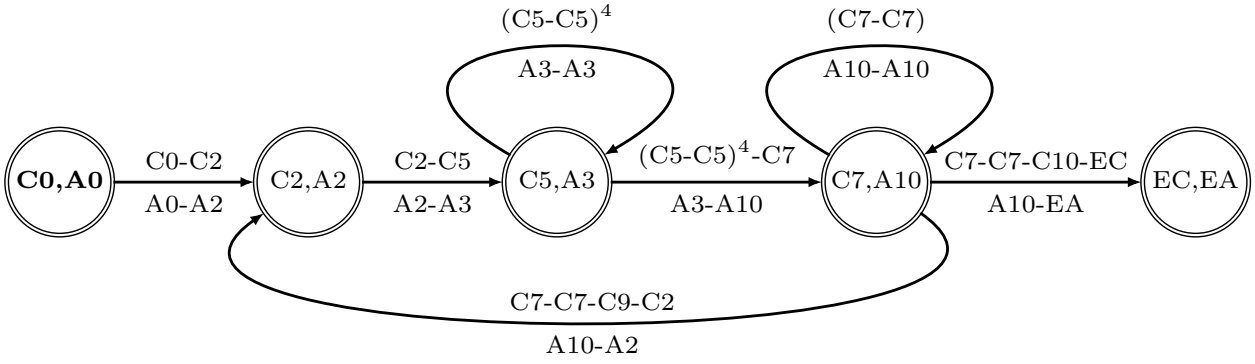
3.2.3 Motivating Example 3

```
int in1[LEN][LEN], in2[LEN];
int out1[LEN], out2[LEN];
C0: void kernel_mvt() {
C1:  int i, j;
C2:  for (i = 0; i < LEN; i++) {
C3:    int sum1 = out1[i];
C4:    int sum2 = out2[i];
C5:    for (j = 0; j < LEN; j++)
C6:      sum1 += in1[i][j] * in2[j];
C7:    for (j = 0; j < LEN; j++)
C8:      sum2 += in1[j][i] * in2[j];
C9:    out1[i] = sum1;
C10:   out2[i] = sum2;
C11:  }
EC: }
```

(a) C program.
LEN is a positive multiple of 4.

```
A0: kernet_mvt:
A1:  r1 = 0
A2:   r2=0; r3=out1[r1]; r4=out2[r1]; xmm0=0
A3:    xmm0 += in1[r1][r2..r2+3]*in2[r2..r2+3]
A4:    r2 += 4
A5:    if (r2 != LEN) goto A3
      // shift right by 8 bytes
A6:   xmm0 += (xmm0 >> 8)
      // shift right by 4 bytes
A7:   xmm0 += (xmm0 >> 4)
A8:   r3 += xmm0[31:0]
A9:   r5 = 0
A10:   r4 += in1[r5][r1] * in2[r5]
A11:   r5++
A12:   if (r5 != LEN) goto A10
A13:  out1[r1] = r3, out2[r1] = r4
A14:  r1++
A15:  if (r1 != LEN) goto A2
EA: ret
```

(b) (Abstracted) Assembly Program



(c) Product-CFG

Figure 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.

The third motivating example discusses the limitation of the SPA approach [11] due to its dependence on the availability of a common AP (alignment predicate) for the complete program. Consider the pair of programs in fig. 3.3 taken from the Polybench suite [57]. The C program contains three loops with two-level maximum nesting depth. Its corresponding assembly program also has three loops where the first inner loop has been unrolled four times.

An ideal AP precisely identifies the correlated transitions; in this example, the ideal AP is different for each loop. For example, the ideal AP for the outer loop may need to relate variables `i` and `r1` while the two inner loops may need to relate variables `j` and `r2`, and `j` and `r5` in their respective APs. However, the SPA algorithm accepts a single AP, which in this case is : $(\texttt{i} = \texttt{r1}) \wedge ((\texttt{j} = \texttt{r2}) \vee (\texttt{j} = \texttt{r5}))$. Such APs are outside the scope of grammar used for synthesizing the AP in the SPA algorithm.

A strong AP which can be synthesized from the grammar used by SPA algorithm would prune out the required PAA, while a weaker AP may result in too many spurious potential PAAs and the algorithm for finding the required PAA from these spurious potential PAAs is unclear. For instance in this example, if the AP is too strong, such as $(\texttt{i} = \texttt{r1}) \wedge (\texttt{j} = \texttt{r2})$, we will miss the required correlation, e.g., we will never be able to correlate the transitions for the PC `(C7,A10)` in the second inner loop. On the other hand, if the AP is too weak, such as $\texttt{H}_\texttt{C} = \texttt{H}_\texttt{A}$ (i.e. the equality of the non-temporary memory state `H`), then we would get a lot of spurious correlations, e.g., a single iteration of the C program's loop would get correlated with a single iteration of the assembly program's loop which would be incorrect.

Churchill et al. acknowledge in their paper [11] that they leave the problem of synthesizing a complex AP for programs with multiple loops for future work. In general to handle programs with multiple loops, the correlation must be developed incrementally: each loop may have its own alignment properties and using a single alignment predicate (AP) for the whole program is unlikely to succeed. The proposed *Counter* algorithm incrementally constructs the product-CFG through counterexample-guided pruning and ranking algorithms, and does not depend on a common alignment predicate for the whole program. The proposed counterexample-guided pruning and ranking algorithms are discussed in more detail in section 3.3.7.

3.3 *Counter* Algorithm

We identify the following improvement opportunities with respect to the state-of-the-art correlation algorithms:

1. An algorithm that needs to identify correlations for each program path individually is not scalable because the number of potential program paths is exponential in program size and unroll factor. To tackle this problem, an algorithm should potentially identify correlation for a set of program paths, or *pathset*, in a single step.
2. In the presence of multiple loops, the correlation must be developed *incrementally*: each loop may have its own alignment properties and using a single alignment predicate (AP) for the whole program is unlikely to succeed in general.
3. An exhaustive search in depth-first order to identify the required product-CFG from the exponentially large search space is not scalable and becomes intractable in the presence of aggressive optimizations that result in significant structural differences between the program-pair. To handle these optimizations, a more robust approach to efficiently find the required product-CFG is required.

We propose a *counterexample-guided best-first search algorithm*, called ***Counter*** to efficiently search the space of potential product-CFGs to yield a provable bisimulation relation.

3.3.1 Top-level Procedure for *Counter* Algorithm

Counter algorithm is based on the incremental correlation approach discussed in section 2.3.2. The pseudo code for the top-level procedure, **bestFirstSearch()**, is shown in fig. 3.4. In this pseudo code, the procedures that are new as compared to the incremental algorithm proposed by Dahiya et al. [14] (fig. 2.5) are shown in boldface.

`bestFirstSearch()` takes as input the CFG for the specification program C and the CFG for the implementation program A. It also takes a parameter μ_C as input which is used to bound the length of the candidate correlations in C. The procedure starts with a *partial product-CFG* π_{init}

Algorithm 2: Top-level Procedure of *Counter* Algorithm

```
1  Function bestFirstSearch(C, A, μ_C)
2      π_init ← initProductCFG(C, A);
3      Ω ← { };
4      expandProductCFG(π_init, C, A, Ω, μ_C);
5      while Ω is not empty do
6          π_cur ← removeMostPromising(Ω);
7          if checkCriterionForNewEdge(π_cur) then
8              inferInvariantsAndCounterExamples(π_cur);
9              if checkCriterionForEdges(π_cur) then
10                 isComplete ← expandProductCFG(π_cur, C, A, Ω, μ_C);
11                 if isComplete ∧ ProductCFGisProvableBisim(π_cur) then
12                     return π_cur;
13                 end
14             end
15         end
16     end
17     return null;
```

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

that has only a single node `(C0,A0)` formed by pairing the start node `C0` in C and the start node `A0` in A. The algorithm then incrementally expands the partial product-CFG π_{init} by adding a new edge to it through a call to `expandProductCFG()` function till a *complete product-CFG* is obtained (line number 4,10 in fig. 3.4). If a product-CFG that is both complete and has strong enough invariants to prove equivalent observable behavior is found, then the top-level procedure returns this product-CFG (along with the invariants) as the bisimulation proof of equivalence across the input specification and implementation CFGs (line number 11-12 in fig. 3.4).

As motivated in section 3.2.1, in order to achieve scalability, the product-CFG constructed using the proposed *Counter* algorithm differs from the product-CFG presented in section 2.3.1 and constructs each product-CFG edge by pairing a pathset (instead of a single path) from the two input CFGs C and A. The `expandProductCFG()` procedure enumerates multiple potential pathset pairs for each product-CFG edge to be added and uses a backtracking search tree to store

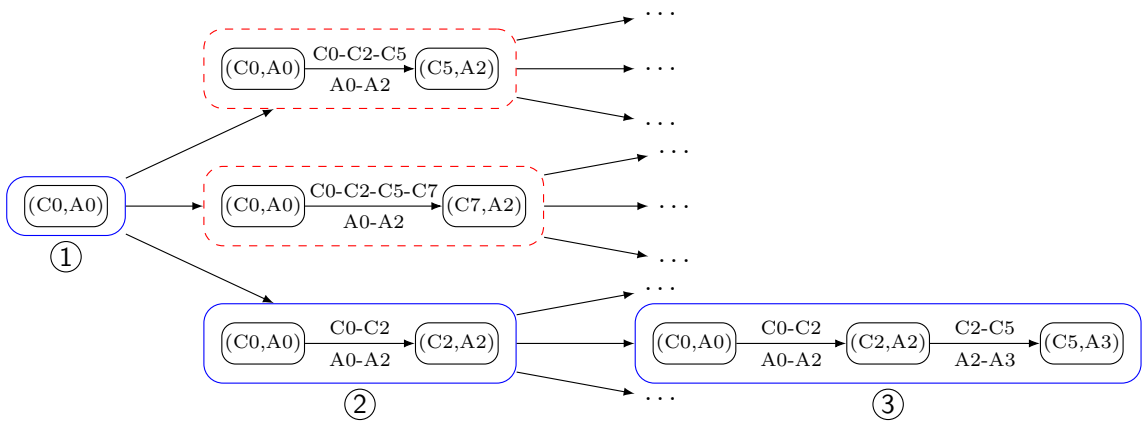


Figure 3.5: A snapshot of the backtracking search tree for the program-pair shown in fig. 3.3

these possible correlations. The top-level procedure initializes the frontier Ω of the backtracking search tree through a call to `expandProductCFG()` function for the initial product-CFG π_{init} (line number 3-4 in fig. 3.4). A snapshot of the search tree for the pair of programs in fig. 3.3 is depicted in fig. 3.5. Each node in the search tree represents a partially-constructed product-CFG, and the outgoing edges at a node in the search tree represent the potential possibilities for the next product-CFG edge to be added.

At each incremental step, unlike prior work that uses a depth-first order to choose a partial product-CFG π_{cur} from the frontier Ω of the backtracking tree, the proposed *Counter* algorithm uses a *best-first search* algorithm to pick the most promising partial product-CFG (`removeMostPromising()`). The proposed best-first search is based on *Counterexample-Guided Pruning and Ranking* algorithm described in section 3.3.7.

After choosing the most procedure correlation π_{cur}, the top-level procedure through the call to `checkCriterionForNewEdge()` function (line number 7 in fig. 3.4) checks that the correlation criterion is met for the new edge in the chosen product-CFG π_{cur} (i.e. the last edge added through the call to `expandProductCFG()` function). Unlike prior work [14], that uses a strong correlation criterion based on equality of correlated paths conditions (or pathset conditions in this context), in section 3.3.5, we propose a more general correlation criterion that can handle a larger space of transformations. Since the correlation criterion is checked individually for each

product-CFG edge being added, the algorithm can thus handle program-transformation pairs that have different alignment properties for different edges in the required product-CFG (for example the program-pair in fig. 3.3).

If the last added edge to the chosen partial product-CFG π_{cur} passes the correlation criterion then inductive invariants are inferred at each node of π_{cur} using an off-the-shelf invariant inference procedure (`inferInvariantsAndCounterExamples()`). The invariant inference procedure used should be powerful enough to find expressive invariants between state elements of programs that have large syntactic gap across them. Simultaneously, the algorithm should be efficient enough to be re-run multiple times at each incremental step for every product-CFG node. We propose such an invariant inference algorithm in section 4.2.

In addition to the inductive node-invariants, *Counter* also requires a set of concrete machine states (data) that may be observed at each product-CFG node for the best-first search. The concrete machine state in the product-CFG is formed by assigning concrete values to the state elements of the abstract machine state for the product-CFG. The machine state of the product-CFG, as detailed in section 2.3.1, is formed by combining the individual abstract machine states of C and A. The *Counter* algorithm does not rely on actual execution traces, but generates these concrete machine states through not-provable SMT queries made during invariant inference (`inferInvariantsAndCounterExamples()`). For example, the potential machine states at the start node `(C0,A0)` may be identified through SMT queries that assert the invariants at that node (e.g., equivalence of input values and non-temporary memory states). Because these concrete states are created from the *models* generated by SMT solver for not-provable queries, we also refer to them as *counterexamples*.

By construction, a counterexample at a product-CFG node must satisfy the inductive invariants at that node. Notice that, both in the presence of UB or without UB, a counterexample need not necessarily be an actually occurring concrete machine state for real inputs because the invariant at a node could be weaker than the strongest set of possible concrete states at a node for real inputs. As long as the counterexamples satisfy the inferred inductive invariants at the respective nodes, using these counterexamples is sound and is precise enough.

After invariant inference, the alignment condition (or the respective correlation criterion) for all edges is checked again using `checkCriterionForEdges()` function, as the conditions may not hold for the updated invariants. If the alignment condition of any of the edge is not satisfied after invariant inference then the algorithm *backtracks* and chooses the next most promising correlation candidate from the frontier (line number 5-6 in fig. 3.4). If the alignment condition still holds for all edges in the partial product-CFG constructed so far (π_{cur}) then the algorithm "expands" this partial product-CFG by adding a new product-CFG edge to it through call to `expandProductCFG()` function (line number 10 in fig. 3.4).

3.3.2 Incremental Procedure to add a new product-CFG edge

At each incremental step of the top-level `bestFirstSearch()` procedure, the **expandProductCFG()** procedure, that takes a partial product-CFG π as input and enumerates multiple pathset-pair candidates for the next edge to be added to it, is called. The pseudo code for the `expandProductCFG()` procedure is shown in fig. 3.6. In this pseudo code, the sub-procedures that are different (or new) as compared to the incremental `expandProductCFG()` procedure (fig. 2.5) used by the prior work [14], are shown in boldface.

The first step in the `expandProductCFG()` procedure is to identify an incomplete PCpair (or node) $n = (n_C, n_A)$ in π through a call to `findIncompleteNode()` function. Section 2.2.2 provides the definition of an incomplete node as a non-exit node in the CFG for which the union of edge-condition of all outgoing edges at that node is not a tautology. In the context of the product-CFG, an edge is formed by correlating pathsets in C and A and a non-exit PCpair is incomplete if the union of pathset-condition of the pathset in A for all outgoing edges at that PCpair is not *true*. This definition of an incomplete non-exit PCpair may not hold for any general product program construction technique and is true only for the lock-step execution based correlation which requires that, for any product-CFG edge $\omega = (\xi_C, \xi_A)$, if any of the paths in ξ_A is traversed in program A, then one of the paths in ξ_C in program C must be traversed. If all the paths are already correlated in the input product-CFG π (i.e. the product-CFG is complete), then the `findIncompleteNode()` procedure returns a `nullptr` and the `expandProductCFG()` func-

Algorithm 3: Incremental Procedure to add a new product-CFG edge

```
1  Function expandProductCFG(π, C, A, Ω, μ_C)
2      if ¬ ((n_C, n_A) ← findIncompleteNode(π)) then
3          return True;
4      end
5      ξ_A ← getNextPathsetRPO(n_A, A);
6      κ_C ← getCandCorrelations(n_C, C, ξ_A, μ_C);
7      foreach ξ_C ∈ κ_C do
8          if actionsAreCompatible(ξ_C,ξ_A) then
9              π_new ← π;
10             addEdgeAndPropCEs(π_new, (ξ_C, ξ_A));
11             if CEsSatisfyCorrelCriterion(π_new) ∧
                InvRelatesMemAtEachNode(π_new) then
12                 Ω ← Ω ∪ {π_new};
13             end
14         end
15     end
16     return False;
```

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

tion returns to the caller which will eventually check if the inferred invariants ensure equivalent observable behavior (`ProductCFGisProvableBisim()`).

If there exists an "incomplete node" (i.e., a node which requires a new correlation for an outgoing pathset in A), $n = (n_C, n_A)$, the algorithm identifies the next pathset ξ_A starting at node n_A in A in *Reverse Post-Order* (RPO) that has not yet been correlated. The algorithm then through a call to `getCandCorrelations()` function enumerates all possible correlations (or pathsets) in C for the chosen pathset ξ_A starting at node n_C in a set of pathsets κ_C. For each of the pathsets $\xi_C \in \kappa_C$, the algorithm first checks if the actions are compatible with ξ_A (`actionsAreCompatible()`), i.e., ξ_c should have exit PC as target iff ξ_A has exit PC as target, as the program exit edge could potentially produce an "observable action". If the action compatibility check passes, a new product-CFG π_{new} is created which additionally includes a new product-CFG edge encoding the new candidate correlation between ξ_C and ξ_A. Counterex-

amples are propagated on the newly added edge (`addEdgeAndPropCEs()`) before applying the pruning criteria: `CEsSatisfyCorrelCriterion()` implements pruning based on paths taken by counterexamples, and `InvRelatesMemAtEachNode()` implements pruning based on memory relations. The newly-created product-CFG π_{new} is added to the frontier (Ω) only if both these subroutines return `true`. Note that the frontier Ω is passed as a *call by reference* parameter to the `expandProductCFG()` procedure.

We next introduce the notion of a pathset formally in section 3.3.3 and present the algorithm to enumerate the candidate pathsets in A and C CFGs in section 3.3.4. The detailed discussion on the `addEdgeAndPropCEs()` function is presented in section 3.3.6. Section 3.3.7 presents the counterexample-guided pruning and ranking algorithm which corresponds to `CEsSatisfyCorrelCriterion()`, `InvRelatesMemAtEachNode()` and `removeMostPromising()` functions in the pseudocode.

3.3.3 Pathset Correlation

A pathset ξ can be thought as a set of paths where a path (as defined in section 2.3.1) is formed by the series composition of edges. The notion of pathset is analogous to the composite path defined in section 2.3.2. A valid pathset used for correlation by the *Counter* algorithm has the following two requirements:

1. All execution paths in ξ start at the same node and end at the same node. For example, in fig. 3.1a, the paths `(C1-C3-C1)` and `(C1-C5-C1)` may be a part of the same pathset; but paths `(C1-C3-C1)` and `(C1-C5-EC)` may not be a part of the same pathset.

2. All execution paths in a pathset should be pairwise mutually-exclusive, i.e., for a given input, if one path in the pathset is taken, then another path in the same pathset cannot be taken simultaneously. In other words, for all possible machine states at the start node of the pathset, the path condition of exactly one of the paths in the pathset can be true.

   As an example, in fig. 3.1a, the paths `(C1-C3-C1)` and `(C1-C5-C1-C3-C1)` may be a part of the same pathset. However, the paths `(C1-C3-C1)` and `(C1-C3-C1-C5-C1)` cannot be a

part of the same pathset because the first path `(C1-C3-C1)` is a prefix of the second path `(C1-C3-C1-C5-C1)` indicating that if the second path is taken, the first path is also taken simultaneously.

We compactly represent a pathset as a *directed series-parallel graph (SP-graph*[1]*)* of individual edges of the CFG. An SP-graph (SPG) is formed using series and parallel composition operators through a context-free grammar as follows:

$$\texttt{SPG ::= } \epsilon \texttt{ | } e \texttt{ | (SPG-SPG) | (SPG + SPG)}$$

Here, ϵ represents the empty path (a path with no edges), and e represents an edge in the CFG. `+` operator indicates parallel composition, and the serial composition is denoted by simply concatenating the edges consecutively using `-` operator in a string. A numeric superscript P^n is used as a shorthand notation for P serially composed with itself n times. For example, $\texttt{(C1-(C2-C3)}^2\texttt{-(}\epsilon\texttt{+C4)-C5)}$ represents a set of two paths, namely `(C1-C2-C3-C2-C3-C5)` and `(C1-C2-C3-C2-C3-C4-C5)`.

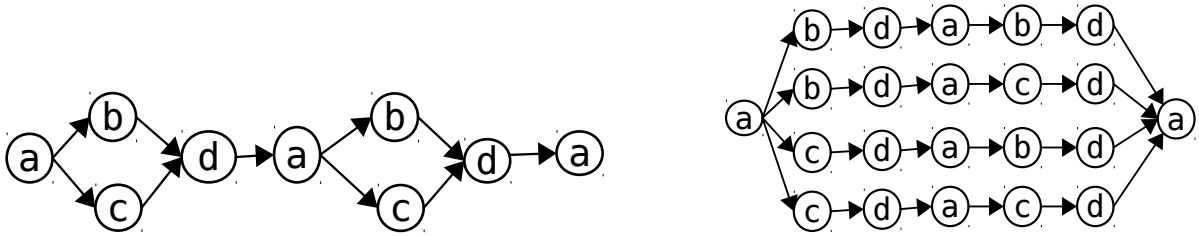
The set of individual paths that are represented in a compact form using the SP-graph can be enumerated through the recursive `enum` procedure as follows:

$$\begin{aligned}
&\texttt{enum}(\epsilon) = \epsilon \quad ; \quad \texttt{enum}(e) = e \\
&\texttt{enum}(spg_1 + spg_2) = \texttt{enum}(spg_1) \cup \texttt{enum}(spg_2) \\
&\texttt{enum}(spg_1 - spg_2) = \chi(\texttt{enum}(spg_1), \texttt{enum}(spg_2))
\end{aligned}$$

where $\chi(x, y)$ forms a Cartesian product of the paths in pathsets x and y, composing each ordered pair in series. As an example, if the pathset x consists of two paths `(a-b-d)` and `(a-c-d)` and the pathset y also consists of two paths `(d-e-g)` and `(d-f-g)` then $\chi(x, y)$ is formed by taking the Cartesian product of paths in the pathsets x and y while composing the paths in series. In this example, $\chi(x, y)$ consists of following four paths: `(a-b-d-e-g)`, `(a-c-d-e-g)`, `(a-b-d-f-g)` and `(a-c-d-f-g)`.

Figure 3.7a shows the SP-graph representation of an example pathset and fig. 3.7b shows the expanded representation formed by enumerating all individual paths for the same pathset obtained

[1]The term SP-graph may suggest that it represents an undirected graph but it is used as a shorthand for "directed series-parallel graph".



(a) SP-graph representation (b) Expanded representation

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

after applying the recursive `enum` procedure. The total number of paths in this example pathset are four, and the length of the longest path is seven.

If $\texttt{enum}^*(\xi)$ represents the set of individual paths obtained after applying the recursive `enum` procedure on a pathset ξ then the two requirements for a well-formed pathset can be formally defined as:

1. All execution paths in ξ start at the same node and end at the same node.

   $\forall\{p_1, p_2\} \in \texttt{enum}^*(\xi) : (\texttt{start}(\texttt{p}_1) = \texttt{start}(\texttt{p}_2) \ \wedge \texttt{end}(\texttt{p}_1) = \texttt{end}(\texttt{p}_2))$

   Here, $\texttt{p}_i$ represents an individual path in the expanded set of paths $\texttt{enum}^*(\xi)$, $\texttt{start}(\texttt{p}_i)$ denotes the start (or source) node of the path and $\texttt{end}(\texttt{p}_i)$ denotes the end (or target) node of the path.

2. All execution paths in a pathset should be pairwise mutually-exclusive.

   $\forall\{p_1, p_2\} \in \texttt{enum}^*(\xi) : (\texttt{pathCond}(\texttt{p}_1) \neq \texttt{pathCond}(\texttt{p}_2))$

   Here, $\texttt{pathCond}(\texttt{p}_i)$ denotes the path condition of $\texttt{p}_i$. As dicussed in section 2.3.1, the path condition is computed as the weakest-precondition of the `True` predicate on the given path.

As motivated in section 3.2.1, an algorithm that needs to identify correlations for each program path individually is not scalable because the number of potential program paths to be correlated can be exponential in program size and unroll factor. To tackle this problem, our proposed correlation algorithm, *Counter*, attempts to correlate a pathset in C with a pathset in A. An

edge $\omega = (\xi_c, \xi_a)$ in the product-CFG constructed using the *Counter* algorithm is formed by pairing pathsets ξ_C and ξ_A in C and A respectively. The lock-step execution represented by a product-CFG edge in the context of pathset is defined as: if the abstract machine traverses the product-CFG edge $\omega = (\xi_c, \xi_a)$, then it would have traversed one of the paths in pathset ξ_C in C *and* it would have traversed one of the paths in pathset ξ_A in A. The number of paths in a pathset ξ for the required product-CFG can potentially be exponential in the size of the program and unroll factor (μ_C). For example, in fig. 3.1, the product-CFG correlates a pathset in C formed by $3^4 = 81$ paths represented as series-parallel digraph `((C1-C3-C1)+(C1-C5-C1)+(C1-C7-C1))`^{4} with pathset `(A2-A2)` (having a single path) in program A. Our decision to allow a single edge in the product-CFG to represent a pair of pathsets and not just single paths in C or A thus reduces the number of edges required in the product-CFG.

3.3.4 Algorithm for Enumerating Candidate Pathsets

For a given partial product-CFG π and an identified incomplete PCpair $n = (n_C, n_A)$ in π, `expandProductCFG()` identifies the potential candidates for the next edge $\omega[n \rightarrow n^d] = (\xi_C, \xi_A)$ to be added. Here ξ_C represents a pathset in C, ξ_A represents a pathset in A and the product-CFG edge ω represents a lock-step execution based correlation across the (ξ_C, ξ_A) pathset-pair.

Anchor Nodes in A and C

To reduce the number of candidate correlations for each product-CFG edge to be added, we restrict the nodes (PCs) of a program that may be correlated in the product-CFG. We call the restricted set of PCs as that program's ***anchor nodes***.

For programs without function calls (as considered in our work), A's anchor nodes are restricted to one of the following possibilities:

1. The start and exit nodes of A.
2. Loop heads in A for loop bodies that do not already contain an anchor node on one of the cyclic paths. We use depth-first search to identify the loop heads on cyclic paths. Considering loop heads as anchor nodes ensures that there is at least one PC in every cyclic

path in A that may get correlated in the product-CFG.

Similarly, for the programs without function calls, the anchor nodes in C include all basic block head and tail nodes in C. Considering all basic block head and tail nodes in C instead of just loop heads as anchor nodes results in robustness and maximizes the probability of finding the required product-CFG with the same invariant inference procedure. Our choice of anchor nodes for A and C appeals to Observation-B in section 3.1, so that we still expect the space of potential product-CFGs to contain the required solution.

In fig. 3.3b, `A0`, `A2`, `A3`, `A10`, and `EA` form the set of anchor nodes for program A. Similarly, in fig. 3.3a, `C0`, `C1`, `C2`, `C3`, `C5`, `C6`, `C7`, `C8`, `C9`, `C11`, and `EC` form the set of anchor nodes in C.

(μ, δ)-Unrolled Pathset

To tackle different possible unrollings, we introduce two parameters *(μ, δ)* to characterize a pathset. A given pathset (or the equivalent set of paths in the expanded representation) from node '`s`' to node '`t`' can be called as a (μ, δ)-*unrolled pathset*, if no node other than `t` is repeated more than μ times on any path in the pathset and `t` is repeated exactly δ times on all paths. We denote a (μ, δ)-*unrolled pathset* as $\texttt{FP}^{\mu,\delta}_{s \rightsquigarrow t}$. Here `FP` is used to represent a *Full Pathset*, i.e. for a given program if there exist a path `P` between `s` and `t` such that no node other than `t` is repeated more than μ times on `P` and `t` is repeated exactly δ times on P, then P must be a part of the enumerated $\texttt{FP}^{\mu,\delta}_{s \rightsquigarrow t}$. For example, in fig. 3.3a, the pathset $\texttt{FP}^{2,2}_{\texttt{C2} \rightsquigarrow \texttt{C7}}$ must contain all three paths `(C2-C5-C7-C7)`, `(C2-C5-C5-C7-C7)`, and `(C2-C5-C7-C2-C5-C7)`,

In our algorithm, we are interested in enumerating all pathsets where $\delta \leq \mu$. To achieve tractability, we use an under-approximate algorithm to construct (μ, δ)-*unrolled pathsets*. In other words, the (μ, δ)-unrolled pathsets enumerated using our algorithm may miss some of the possible paths in the program that have no node repeated more than μ times and t is repeated exactly δ times on those paths. We observe that it is very rare that the paths not enumerated by our algorithm in the pathsets are relevant for correlation across compiler transformations and have not come across any instance yet. We also define $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t} = \{\texttt{FP}^{\mu,\delta}_{s \rightsquigarrow t} | 1 \leq \delta \leq \mu\}$. The maximum number of pathsets in $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t}$ can be μ, where each element in the set may represent the pathset $\texttt{FP}^{\mu,i}_{s \rightsquigarrow t}$ (for $1 \leq i \leq \mu$).

Algorithm 4: Algorithm for *(μ, δ)-unrolled pathset* enumeration

```
1  Function getUnrolledSubgraph(s, G, μ, vMap)
2      subGraph ← getUnrollOneSubgraph(s, G, vMap, μ);
3      if (¬ Edges(subGraph)) then
4          return { };
5      end
6      if (s_new ← chooseMostPromisingSink(subGraph)) then
7          vMap_new ← updateVistedMap(vMap, G, subGraph, s_new);
8          if (vMap_new(n) ≤ μ, ∀n ∈ Nodes(G)) then
9              subGraph ⊙ getFullPathset(s_new, G, μ, vMap_new);
10         end
11     end
12     return subGraph;
13 Function getPathsetAtAllDeltas(s, t, G, μ)
14     vMap ← {n ← 0, ∀n ∈ Nodes(G) };
15     unrolledG ← getUnrolledSubgraph(s, G, μ, vMap);
16     pathsets ← SPgraphReduction(unrolledG, t);
17     return pathsets;
```

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

In fig. 3.3a, $\texttt{FPsets}^{2}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$ is a set of two elements: $\{\texttt{FP}^{2,1}_{\texttt{C5}\rightsquigarrow\texttt{C7}}, \texttt{FP}^{2,2}_{\texttt{C5}\rightsquigarrow\texttt{C7}}\}$. Here, $\texttt{FP}^{2,1}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$ contains `(C5-C7)`, `(C5-C5-C7)`, and `(C5-C5-C5-C7)`. Similarly, $\texttt{FP}^{2,2}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$ contains `(C5-C7-C7)`, `(C5-C5-C7-C7)`, `(C5-C5-C5-C7-C7)`, `(C5-C7-C2-C5-C7)`, `(C5-C5-C7-C2-C5-C7)`, and `(C5-C7--C2-C5-C5-C7)`. Note that $\texttt{FP}^{2,1}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$ and $\texttt{FP}^{2,2}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$ are disjoint (they do not have a common path). In general, two full pathsets with different values of δ are disjoint by definition. Further, all paths within a single pathset (say $\texttt{FP}^{2,2}_{\texttt{C5}\rightsquigarrow\texttt{C7}}$) are pairwise mutually-exclusive by construction.

Algorithm to enumerate (μ, δ)-Unrolled Pathsets

Figure 3.8 shows the pseudo code for the algorithm used to enumerate *(μ, δ)-unrolled pathsets* starting at node `s` and ending at node `t` in CFG G for a given unroll factor μ. The top-level procedure **getPathsetAtAllDeltas()** computes the set of *(μ, δ)-unrolled pathsets* i.e. $\texttt{FPsets}^{\mu}_{s\rightsquigarrow t}$ where each individual pathset in this returned set represents $\text{FP}^{\mu,\delta}_{s\rightsquigarrow t}$, where $1 \leq \delta \leq \mu$. This top-

level function starts by initializing a visited count map `vMap` to `0` for all PCs (aka nodes) in the input CFG G. It passes this visited count map `vMap` along with the start node `s`, the CFG G, and the unroll factor μ to a recursive helper function **getUnrolledSubgraph()**.

The pseudo code for the helper function `getUnrolledSubgraph()` is shown in fig. 3.8. The helper function starts by constructing a subgraph of the CFG (`subGraph`) rooted at PC '`s`' through a call to `getUnrollOneSubgraph()` function. While constructing this subgraph, `getUnrollOneSubgraph()` function only adds those edges for which the target node is reachable from the node '`s`' and its visited count (stored in `vMap`) is less than the input unroll factor μ. Further, the subgraph is constructed as a directed-acyclic-graph (DAG) by adding multiple versions of the same PC belonging to the original CFG, one for each unrolled iteration. If the subgraph returned by `getUnrollOneSubgraph()` function does not have any edges then either all the reachable PCs from node s have been visited μ times already or a terminal node (i.e. exit node) is reached. The recursive function returns with an empty subgraph in this case.

If the subgraph returned by `getUnrollOneSubgraph()` function (`subGraph`) has non-zero edges, then the helper function proceeds by heuristically choosing the "most promising sink" in this subgraph (`chooseMostPromisingSink()`). This is the node which we expect to dominate the unrolled iterations (i.e., all unrolled iterations must go through the chosen most promising sink). Based on the heuristic, if an incorrect most promising sink is selected by the algorithm, then the desired correlation candidate which can potentially yield a bisimulation proof may not get enumerated. For most program-transformation pairs, we use the heuristic that chooses the head of the loop that is being unrolled as the most promising sink.

For each PC, that appears as a target node in an edge in `subGraph` and the chosen sink node is reachable from that PC, the value in the visited count map ($\texttt{vMap}_{\texttt{new}}$) is incremented by one. If the value in the updated visited count map $\texttt{vMap}_{\texttt{new}}$ is greater than the unroll factor μ for any PC, then the function returns the `subGraph` constructed so far. Otherwise, the `getUnrolledSubgraph()` function make a recursive call to itself with the chosen (most promising) sink as the new '`s`' and the updated visited count map $\texttt{vMap}_{\texttt{new}}$ as arguments. The returned graph from the recursive call is rooted at the chosen sink $\texttt{s}_{\texttt{new}}$ and is appended to the `subGraph` enumerated so far before returning it.

The unrolled subgraph (`unrolledG`) returned by `getUnrolledSubgraph()` function is a general DAG (directed acyclic graph) and a a series-parallel decomposition (i.e. a set of SP-graphs) is constructed from it through a call to **SPgraphReduction()** function. A well-formed SP-graph is a directed graph that can be formed using the context-free grammar based on series and parallel composition operators introduced in section 3.3.3. We adapt the reduction algorithm presented in [70] for generating the set of well-formed SP-graphs from a given DAG. When the input DAG is not SP-reducible (i.e. cannot be reduced to set of SP-graphs), we duplicate nodes to make it suitable for the reduction procedure (§9.6.4 in [2]). We use heuristics to minimize the duplication of nodes and call the resulting SP-graph, a *Minimal Series-Parallel Digraph* (MSP).

The MSP representation enables linear-sized SMT proof obligations while determining correlations across pathsets, even when a single pathset may contain an exponential number of individual paths. For example, the MSP representation of the pathset from `(C2-C2)` i.e. `((C1-C3-C1)+(C1-C5-C1)+(C1-C7-C1))`^{4} shown in fig. 3.1c results in the following linear-sized expression for the memory during the proof obligation:

$$
\begin{aligned}
&\texttt{data1} = \texttt{a[i]} + \texttt{ite}(\texttt{d[i]} < 0, \texttt{b[i]} * \texttt{c[i]}, \texttt{ite}(\texttt{d[i]} == 0, \texttt{b[i]} * \texttt{b[i]}, \texttt{c[i]} * \texttt{c[i]}))\\
&\texttt{data2} = \texttt{a[i+1]} + \texttt{ite}(\texttt{d[i+1]} < 0, \texttt{b[i+1]} * \texttt{c[i+1]}, \texttt{ite}(\texttt{d[i+1]} == 0, \texttt{b[i+1]} * \texttt{b[i+1]}, \texttt{c[i+1]} * \texttt{c[i+1]}))\\
&\texttt{data3} = \texttt{a[i+2]} + \texttt{ite}(\texttt{d[i+2]} < 0, \texttt{b[i+2]} * \texttt{c[i+2]}, \texttt{ite}(\texttt{d[i+2]} == 0, \texttt{b[i+2]} * \texttt{b[i+2]}, \texttt{c[i+2]} * \texttt{c[i+2]}))\\
&\texttt{data4} = \texttt{a[i+3]} + \texttt{ite}(\texttt{d[i+3]} < 0, \texttt{b[i+3]} * \texttt{c[i+3]}, \texttt{ite}(\texttt{d[i+3]} == 0, \texttt{b[i+3]} * \texttt{b[i+3]}, \texttt{c[i+3]} * \texttt{c[i+3]}))\\
&\texttt{mem\_out} = \texttt{store}(\texttt{store}(\texttt{store}(\texttt{store}(\texttt{mem}, \texttt{a[i]}, \texttt{data1}), \texttt{a[i+1]}, \texttt{data2}), \texttt{a[i+2]}, \texttt{data3}), \texttt{a[i+3]}, \texttt{data4})
\end{aligned}
\tag{3.1}
$$

The discharge of these linear-sized SMT proof obligations generated due to the control-flow transformations performed by a typical compiler, are usually fast and are slightly slower than the time taken for smaller queries generated while correlating each individual path separately. Also, even though, the discharge of linear-sized SMT proof obligations generated for the complete pathset (having exponential number of paths) remains worst-case exponential-time, an algorithm that attempts to correlate each individual path separately would require an exponential amount of time for an exponential number of paths, even for computing equivalence across trivial transformations.

In give an empirical comparison, the above shown linear-sized proof obligation (eq. (3.1)) for the memory for the example program in fig. 3.1c (with 4 unrolling) is `4x` slower than the average time

taken by the smaller queries generated while correlating each individual path separately. But, since an exponential number of queries ($3^4 = 81$) are generated while correlating each individual path separately, the total time taken is appox. `20` times more than the time taken to discharge the linear-sized proof obligation for the MSP representation of the pathset. Also, the time taken to discharge the above shown linear-sized proof obligation (eq. (3.1)) is only `1.25x` slower than the maximum time taken by any of the smaller queries generated while correlating each individual path separately.

The reduction algorithm adapted from [70] involves an outer loop that performs one of the following reduction operations (in the same precedence order) repeatedly until the entire graph reduces to a single edge. For this reduction procedure, we associate each node of the DAG with all SP-graphs that have been collected as sink starting at that node. Further, we associate with each edge of the DAG an SP-graph to each of the "reduced PCs", i.e. nodes that were reduced (or eliminated) while constructing this edge. Further, the edge is also associated with the SP-graph to the target node of the edge.

Reduction Operations:

1. **Parallel Reduction:** Merge two edges that have the same source and target nodes to result in a single edge. The new edge is associated with the SP-graphs (one per reduced node or target node) formed by the parallel composition of the SP-graphs associated with the two edges with the same reduced or target node
   PARALLEL(ω_1[a→b], ω_2[a→b]) $\implies$ (ω_1 + ω_2)[a→b];
   $\forall$ n $\in$ {$R_{\omega 1}$, $R_{\omega 2}$, b} : PARALLEL(SPG$_1$(a→n), SPG$_2$(a→n)) $\implies$ (SPG$_1$ + SPG$_2$)(a→n)
   Here R_ω represents the set of reduced PCs associated with the edge ω.

2. **Series Reduction:** Merge two edges composed in series, where the common node has only one incoming and one outgoing edge, into a single edge. The SP-graphs associated with the new edge are formed by the union of SP-graphs at the first edge and the serial composition of the SP-graph from source to target of the first edge with the SP-graphs associated with the common node and the SP-graphs associated with the second edge. Here, union can be thought as the application of the parallel reduction operation on the resulting SP-graphs after the series composition.

SERIES(ω_1[a→b], ω_2[b→c]) $\implies$ (ω_1 ω_2)[a→c];
SERIES(SPG$_1$(a→b), SPG$_2$(b)) $\implies$ (SPG$_1$ SPG$_2$)(a→b)
$\forall$ n $\in$ {R$_{\omega 2}$, c} : SERIES(SPG$_1$(a→b), SPG$_2$(b→n)) $\implies$ (SPG$_1$ SPG$_2$)(a→n)
$\forall$ n $\in$ {R$_{\omega 1}$, R$_{\omega 2}$, b, c} : UNION(SPG$_1$(a→n), SPG$_2$(a→n)) $\implies$ (SPG$_1$ + SPG$_2$)(a→n)

3. **Reduction of sink node with only one incoming edge:** Collect a "sink" (i.e., a node with no outgoing edges) with only one incoming edge, which involves removing the only incoming edge ($\omega_{ic}[n_{from} \rightarrow n_{sink}]$) to the sink node and removing the sink node itself. If ω_{ic} is also the only outgoing edge from the source node n_{from}, then the SP-graphs associated with n_{from} are updated by the union of original SP-graphs associated with n_{from}, SP-graphs associated with the incoming edge (ω_{ic}) and the application of series reduction on the SP-graphs at the sink node n_{sink} and the SP-graph at ω_{ic} corresponding to n_{sink} PC.

   If n_{from} has another outgoing edge except ω_{ic}, then reduction operation instead of modifying the SP-graphs associated with the from PC n_{from}, updates the SP-graphs associated with the other outgoing edge.

4. **Reduction of sink node with more than one incoming edge:** This case occurs when the input DAG is not SP-reducible and requires duplication of nodes and edges to make it suitable for the reduction procedure. Although most high-level languages (including C-language) encourage structured control flow (for example, by avoiding use of `goto` statement) and thus the graphs generated from such programs are SP-reducible. However, the compiled programs in assembly are often unstructured and may have arbitrary control flow; the graphs generated from most of these assembly programs are thus not SP-reducible.

   The reduction step in this case involves collecting a "sink" (i.e., a node with no outgoing edges) by removing one of the multiple incoming edges to it ($\omega_{ic}[n_{from} \rightarrow n_{sink}]$). The required duplication of nodes and edges involves taking the series composition of the SP-graph for the sink node (n_{sink}) associated with the edge ω_{ic} with each SP-graph associated with the sink node. The SP-graphs associated with the edge to be removed (i.e. ω_{ic}) are then modified by taking their union (i.e. parallel reduction) with the SP-graphs resulting from the series composition.

   Rest of the steps involved in this reduction operation are same as the above reduction operation with only one incoming edge. The only difference is that the modified SP-graphs

Algorithm 5: Algorithm for enumerating the candidate pathset ξ_A in A

```
1 Function getNextPathsetRPO((n_C, n_A), A, π)
2   foreach h_A ∈ nexthops of n_A in A in RPO do
3     ξ_A ← getPathsetAtAllDeltas(n_A, h_A, A, 1);
4     if notAlreadyCorrelated(π, (n_C, n_A), ξ_A) then
5       return eliminatePathsWithOtherAnchorNodes(ξ_A);
6     end
7   end
8   NotReached();
```

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

as discussed here are used during the reduction operation and the sink node n_{sink} is not not removed in this case.

The fixed-point loop of reduction operations results in a graph with single edge and the required *(μ, δ)-unrolled pathsets* $\texttt{FPsets}^{\mu}_{s \rightsquigarrow t} = \{\texttt{FP}^{\mu,\delta}_{s \rightsquigarrow t} | 1 \leq \delta \leq \mu\}$ are formed by the union of the SP-graphs at the only remaining edge, the remaining edge's from-node, and the remaining edge's to-node.

Enumerating the possible pathset-pairs (ξ_C, ξ_A) from C and A CFGs

For an incomplete product-CFG node $n = (n_C, n_A)$ the `expandProductCFG()` function proceeds by choosing a pathset ξ_A in A through a call to **getNextPathsetRPO()** function. The enumerated pathset ξ_A starts at the chosen incomplete node n_A in A and the possible target nodes for ξ_A are restricted to *nexthop anchor nodes* of n_A. A nexthop anchor node of n_A is an anchor node which can be reached from n_A through a program path in A without having to go through any other anchor node in A.

The pseudo code for the `getNextPathsetRPO()` function is shown in fig. 3.9 and involves the following two steps:

1. For each nexthop anchor node h_A of n_A, we compute $\texttt{FPsets}^{1}_{n_A \rightsquigarrow h_A}$ (i.e., unroll factor = 1, indicating no unrolling) through a call to `getPathsetAtAllDeltas()` function. Notice that for a fixed h_A, $\texttt{FPsets}^{1}_{n_A \rightsquigarrow h_A}$ will be a singleton set whose only element repre-

Algorithm 6: Algorithm for enumerating the candidate pathsets ξ_C in C

```
1 Function getCandCorrelations(n_C, C, ξ_A, μ_C)
2 |  κ_C ← {ϵ};
3 |  foreach w_C ∈ C do
4 |  |  κ_C ← κ_C ∪ getFullPathsetAtAllDeltas(n_C, w_C, C, μ_C);
5 |  end
6 |  return κ_C;
```

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

sents the full pathset $\mathrm{FP}^{1,1}_{n_A \rightsquigarrow h_A}$. For example, in fig. 3.3b, $\mathtt{FP}^{1,1}_{\mathtt{A3} \rightsquigarrow \mathtt{A3}}$ can be represented as `((A3-A3)+(A3-A10-A2-A3))`.

2. As a second step, the function `eliminatePathsWithOtherAnchorNodes()` removes those paths from $\mathrm{FP}^{1,1}_{n_A \rightsquigarrow h_A}$ that contain edges incident to any other anchor node in A (except h_A). The resulting pathset ξ_A is the correlation candidate for the next edge to be added to the partial product-CFG constructed so far. For example, in fig. 3.3b, $\mathtt{FP}^{1,1}_{\mathtt{A3} \rightsquigarrow \mathtt{A3}}$ can be represented as `((A3-A3)+(A3-A10-A2-A3))`. After the second step, ξ_A reduces to only `(A3-A3)` and the other path incident to `A3` through `A10` is removed.

By construction all candidate pathsets ξ_A for all nexthop anchor nodes h_A in A are mutually exclusive and the complete product-CFG should have correlations for all such ξ_A pathsets. Our algorithm proceeds by correlating one such ξ_A that is not already correlated in the input partial product-CFG π at each step. To reduce backtracking, we pick these nexthop nodes h_A (and associated candidate pathsets) in *Reverse Post-Order* (RPO); choosing nexthop nodes in RPO order maximizes the likelihood that a node is correlated only after its predecessor nodes are already correlated. For example, in fig. 3.3, starting at product-CFG node `(C5,A3)`, the nexthop anchor nodes in program A in reverse post-order are `A3` and `A10`.

After choosing a pathset ξ_A in A, the `expandProductCFG()` procedure, through a call to `getCandCorrelations()`, enumerates the candidate pathsets in C starting at node n_C. The pseudo code for the **getCandCorrelations()** function is shown in fig. 3.10. It identifies $\mathtt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C}$ for all anchor nodes w_C in C that are reachable from node n_C. It adds all such

pathsets to an accumulator κ_C, i.e., $\kappa_C = \{\epsilon\} \cup (\bigcup_{w_C} \texttt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C})$ (ϵ represents the empty path). Each non-ϵ element in κ_C represents a *(μ, δ)-unrolled pathset* from node n_C to an anchor node w_C in C. Notice that unlike the pathsets enumerated for A, here we do not restrict the anchor node w_C to be the nexthop of n_C.

Consider the example specification program in fig. 3.3a and assume that we are enumerating pathsets in C starting at node `C5` for $\mu_C = 2$. Based on the procedure described in this section, we would consider all reachable anchor nodes from `C5` as w_C, which are `C2`, `C3`, `C5`, `C6`, `C7`, `C8`, `C9`, `C11`, and `EC`. For each possible w_C, the $\texttt{FPsets}^{\mu_C}_{n_C \rightsquigarrow w_C}$ is computed through call to `getPathsetAtAllDeltas()` function.

For each of the pathsets $\xi_C \in \kappa_C$, the `expandProductCFG()` function checks if the actions are compatible with ξ_A (`actionsAreCompatible()`), i.e., ξ_A should have exit PC as target iff ξ_C has exit PC as target, as the program exit edge could potentially produce an "observable action". If the action compatibility check passes, a new product-CFG π_{new} is created which additionally includes a new product-CFG edge encoding the candidate correlation between ξ_C and ξ_A.

3.3.5 Criterion for Correlating Pathsets

By definition, if an edge $\omega = (\xi_C, \xi_A)$ is traversed in the product-CFG, it implies that one of the paths in pathset ξ_C is traversed in program C and one of the paths in the pathset ξ_A is traversed in program A. While this property defines the general space of potential correlations, we restrict this space further through a *correlation criterion* to achieve better tractability. We restrict correlations by requiring that ξ_C can be correlated with ξ_A through a product-CFG edge $\omega = (\xi_C, \xi_A)$ only if the following property holds: *if any of the paths in ξ_A is traversed in program A, then one of the paths in ξ_C in program C must be traversed.* For example, our algorithm does not allow $\xi_A =$`(A2-A2)` to be correlated with $\xi_C =$`(C1-C3-C1)`^{4} for the pair of programs in fig. 3.1 because if ξ_A is traversed in A, then paths outside ξ_C may be traversed in C (e.g., `(C1-C5-C1)`^{4}).

This restriction that mandates that ξ_A must represent a specialization of ξ_C is a direct reflection of Observation-C introduced in section 3.1. We argue, through empirical evaluation, that this

restriction significantly reduces the product-CFG search space without hurting robustness.

To compare this restriction with previous work, the SPA algorithm [11] has no such requirement: given the required AP (alignment predicate), it could potentially correlate any path in C with any path in A theoretically. However, for the APs and the transformations considered in their paper, there is no program-pair and associated PAA (or product-CFG) that violates this requirement. This is unsurprising because 27 out of 28 programs evaluated in their work have a single loop in the source program and none of them have control flow within the loop body. This restriction on the other hand allows us to scale to larger programs, while still retaining robustness to a very high degree.

As we discuss later in section 5.3, while this restriction may preclude certain code size transformations, it caters to almost all runtime optimizations that do not attempt to minimize code size. We leave the development of scalable algorithms that overcome this restriction for future work.

3.3.6 Counterexample Propagation

A set of counterexamples (Γ) is associated with each node in the product-CFG and includes the models (or concrete assignments) generated by SMT solver for not-provable queries during invariant inference. As noted in section 2.4, even though counterexamples cannot replace real execution traces, a counterexample is still useful because it satisfies the inferred invariant at the node at which it was generated, *and* it does not trigger UB on the paths on which it is propagated. Thus it is safe to consider a counterexample at par with a real concrete machine state for these smaller program segments where it does not trigger UB, because our reasoning power in these smaller segments is constrained by the inferred invariants in any case.

Counter relies on the counterexamples in this set to prune the infeasible correlation candidates (ξ_C, ξ_A) and to rank the remaining candidates resulting in a best-first search procedure that can scalably handle aggressive transformations. Since, more counterexamples at each product-CFG node are desirable for a more effective pruning and ranking, we additionally *propagate* (or execute) every generated counterexample in the product-CFG in the forward direction to

populate the set of counterexamples at downstream nodes. To ensure termination, we bound the maximum number of times a counterexample propagation may encounter a node — we call this the *propagation bound* and use 3 as the default propagation bound in our algorithm. If the counterexample visits a node more than thrice during propagation, we do not propagate it any further. Using a propagation bound of 3 is meaningful because it is small enough to result in efficient propagations, and yet it is usually able to produce new useful counterexamples from an existing counterexample (e.g., over a loop edge) without requiring more SMT queries.

Given a pathset pair (ξ_C, ξ_A) and a partial product-CFG π_{new}, the **addEdgeAndPropCEs()** function adds a new product-CFG edge encoding the new candidate correlation between ξ_C and ξ_A and propagates the counterexamples at the source node of the new edge to the target node and further (subject to propagation bound). It additionally calls the invariant inference procedure to update the invariant at the target node of the edge to the strongest invariant cover of the counterexample set obtained after propagation at the target node of the new edge. The strongest invariant cover of a given counterexample set is the strongest invariant such that all concrete states present in the counterexample set satisfy the invariant. Notice that this strongest invariant cover for a counterexample set need not be inductive across the newly added edge, but serves as a lower bound on the inductive invariant cover (which would be computed later by the call to `inferInvariantsAndCounterExamples()`). A more detailed discussion on the strongest invariant cover of a given counterexample set is presented in section 4.2.1. Counterexamples also get generated during the invariant inference procedure for not provable SMT queries (`inferInvariantsAndCounterExamples()`). Counterexample propagation (subject to propagation bound) is attempted for these counterexample as well and the strongest invariant cover is recomputed for the nodes for which the set of counterexamples has changed after propagation.

In the next section (section 3.3.7), we demonstrate that the strongest invariant cover for the propagated counterexample set across the newly added edge is helpful in pruning and ranking the candidate partial product-CFGs to implement a best-first exploration of the search space. It is important to note that the computation of the invariant cover is significantly cheaper than an SMT query; thus pruning based on such computation instead of an SMT query is meaningful.

3.3.7 Counterexample-Guided Pruning and Ranking

At every incremental step, the `expandProductCFG()` function, for a chosen partial product-CFG, enumerates multiple candidate correlations for the new edge to be added by pairing an outgoing pathset in A with one of the candidate pathset in C. These correlation candidates are added back to the frontier of the backtracking search tree. To keep the backtracking search tractable, it is crucial to carefully prioritize more promising correlations over others. This prioritization allows us to realize a *best-first search*.

At a high level, we first enumerate all possibilities for pathsets in C (in an arbitrary order) using the `getCandCorrelations()` function. We then *prune* out certain possibilities through call to `CEsSatisfyCorrelCriterion()` and `InvRelatesMemAtEachNode()` functions because the current set of counterexamples decisively indicate that those possibilities cannot yield a provable bisimulation under the constraints of the invariant inference algorithm used. Next, we use the counterexamples to *rank* the remaining possibilities from most promising to least promising.

For the following discussion on counterexample-guided pruning and ranking strategy in this section, we consider an incomplete PCpair $n = (n_C, n_A)$ in the product-CFG and the new product-CFG edge $\omega[n \to n^d]$ represents the candidate correlation added to the given product-CFG from the incomplete node n. This newly-created edge $\omega[n \to n^d] = (\xi_C, \xi_A)$ is formed by correlating the pathset ξ_C (starting at node n_C in C) with the pathset ξ_A (starting at node n_A in A). The counterexample set at node n is represented by Γ_n.

Further, counterexamples in Γ_n are propagated across $\omega[n \to n^d]$ to add counterexamples at node n^d, and potentially n^d's successors, subject to the propagation bound. During counterexample propagation, as the counterexample sets are updated, the strongest invariant covers at n^d (and other downstream nodes) are recomputed. In our following discussion on the ranking procedure, we assume that we are comparing two partial product-CFGs π_1 and π_2 which are otherwise identical, but differ in the most-recently added ω edge. We later generalize this discussion to comparison between arbitrary partial product-CFGs.

Pruning Based on Paths Taken by Counterexamples

If there exists a counterexample $\gamma \in \Gamma_n$ such that, when propagated on the outgoing paths, γ takes a path in ξ_A but does not take any of the paths in ξ_C, then that candidate correlation is discarded. This pruning strategy appeals to our correlation criterion (section 3.3.5) which requires that if program A takes a path in ξ_A, then program C must take one of the paths in ξ_C. If an existing counterexample violates this criterion for a candidate correlation, that candidate correlation is evidently incorrect. The `CEsSatisfyCorrelCriterion()` function implements this pruning strategy and checks the following condition for every product-CFG edge $\omega[n \rightarrow n^d] = (\xi_C, \xi_A)$:

$$Inv_n \Rightarrow (pscond_{\xi_A} \rightarrow pscond_{\xi_C})$$

where $pscond_\xi$ represents the *pathset condition* of pathset ξ, which is equivalent to the disjunction of the path-conditions (i.e., the weakest condition under which the path must be taken) of the individual paths in ξ.

Pruning Based on Memory Relations

If the non-temporary memory states $\mathtt{H_C}$ and $\mathtt{H_A}$ are not related by the (re)computed invariant cover at n^d (or any other downstream node), we eliminate that candidate product-CFG.

This is based on the premise that the memory states need to be correlated at program exit, and if they are not correlated at an intermediate PCpair, then there is little hope for them to be correlated at exit. Thus it is safe to eliminate all such partial product-CFGs. The `InvRelatesMemAtEachNode()` function implements this pruning strategy.

Ranking on Number of Affine-Related Live Bitvector Variables in A

To compare two candidate correlations, represented as two candidate partial product-CFGs, π_1 and π_2, we count the number of live bitvector variables in A that are related through affine (or linear) relations at n^d. If the number of live bitvector variables in program A related through affine relations in $\mathtt{INV}_{n^d}$ is more in π_1 than in π_2, then π_1 is ranked higher than π_2, and vice-versa.

Notice that n^d may be different in π_1 and π_2.

This ranking strategy is based on the heuristic that an incorrect correlation would likely cause some of the live bitvector variables in A to not have any relation to the program values at the correlated PC in C.

Ranking on Number of Affine-Related Live Bitvector Variables in C

If the first ranking step results in a tie (i.e., the number of affine-related live bitvector variables in A is identical in both correlations), we compare the number of live bitvector variables in program C that are related through affine relations in $\texttt{INV}_{n^d}$: if the number of live bitvector variables in C with affine relations is more in π_1, then π_1 is ranked higher, and vice-versa.

This ranking strategy is also based on the heuristic that a correct correlation is likely to relate more C variables than an incorrect correlation.

Static Heuristic as Tie-Breaker

If both correlations behave identically on the two ranking criteria listed above, we use the following *static heuristic* as a tie-breaker:

- Recall that all correlated pathsets in C are (μ, δ)-unrolled full pathsets. Also recall that for the pathset ξ_C between nodes n_C and n_C^d, the node n_C^d is repeated exactly δ times on all paths (indicating δ unrollings). We prioritize the correlation that correlates a pathset with a lower value of δ.

  This heuristic is based on the observation that most program transformations do not involve unrolling, and so it is more efficient on average to prioritize correlations at smaller unroll factors.

- If the unroll factors of the two candidate correlated pathsets ξ_C are identical, then we prioritize the correlation that has ξ_C with a longer *pathset length*. The pathset length is the length of the shortest path in that pathset. This tie-breaker is meaningful because longer paths would generally entail stronger path conditions and thus have a higher likelihood of

Function *computeRank*(π)
  $\mathtt{r}^A \leftarrow 0$ $\mathtt{r}^C \leftarrow 0$
  foreach node $n = (n_C, n_A) \in \mathtt{nodes}(\pi)$ **do**
    $\mathtt{live}^A \leftarrow$ *getLiveVariables*(n_A, A) $\mathtt{live}^C \leftarrow$ *getLiveVariables*(n_C, C)
    $\mathtt{r}^A \leftarrow \mathtt{r}^A +$ *getVariablesWithNoAffineRelations*($\mathtt{live}^A$, $\mathtt{INV}_n$)
    $\mathtt{r}^C \leftarrow \mathtt{r}^C +$ *getVariablesWithNoAffineRelations*($\mathtt{live}^C$, $\mathtt{INV}_n$)
  end
  return (r^A, r^C)
Function *staticHeuristic*(π_1, π_2)
  $\xi_1^C \leftarrow$ *lastCorrelatedSrcPathset*(π_1) $\xi_2^C \leftarrow$ *lastCorrelatedSrcPathset*(π_2)
  $\mu_1 \leftarrow$ *getUnrollFactorForPathset*(ξ_1^C) $\mu_2 \leftarrow$ *getUnrollFactorForPathset*(ξ_2^C)
  $\mathtt{len}_1 \leftarrow$ *getPathSetLength*(ξ_1^C) $\mathtt{len}_2 \leftarrow$ *getPathSetLength*(ξ_2^C)
  return $(\mu_1, -\mathtt{len}_1) \leq (\mu_2, -\mathtt{len}_2)$
Function *comparePromiseForProductCFGs*(π_1, π_2)
  $(r_1^A, r_1^C) \leftarrow$ computeRank(π_1) $(r_2^A, r_2^C) \leftarrow$ computeRank(π_2)
  if $(r_1^A, r_1^C) \neq (r_2^A, r_2^C)$ **then**
    return $(r_1^A, r_1^C) < (r_2^A, r_2^C)$
  end
  return *staticHeuristic*(π_1, π_2)

Figure 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.

failing our correlation criterion (described in section 3.3.5) in case of an incorrect correlation. In other words, this heuristic of prioritizing the longer path over shorter paths resembles a "fail fast" strategy.

So far, we have described the ranking strategy in the context of a single correlation of ξ_A with ξ_C. However, a product-CFG is made up of multiple edges, each denoting a separate correlation. For our best-first search algorithm, we need to compare one product-CFG with another even if they may involve multiple different correlations. To allow such comparisons, we extend these ideas to the whole product-CFG by accumulating the number of live bitvector variables (in A and C) that have not been correlated at every PCpair, and then comparing these accumulated counts. This comparison function is shown in fig. 3.11.

The `comparePromiseForProductCFGs()` function compares two product-CFGs for their relative promise towards yielding a provable bisimulation. This function returns true iff π_1

holds more promise than π_2. This `comparePromiseForProductCFGs()` function is used by the `removeMostPromising()` procedure to choose the most promising partial product-CFG from the frontier of the backtracking tree.

3.3.8 Pruning and Ranking Algorithms Through Examples

Consider the example program-pair in fig. 3.3. Suppose we are considering the partial product-graph ③ in the backtracking search tree for this example program-pair shown in fig. 3.5. The algorithm would next try to correlate the pathset ξ_A = `(A3-A3)` in A starting at the incomplete-node (n_C, n_A) = `(C5,A3)`, based on RPO.

For $\mu_C = 4$, the candidate pathsets (κ_C) enumerated for C would include the ϵ path and *(μ, δ)-unrolled pathsets* starting from n_C = `C5` and ending at one of the following nine reachable anchor nodes in C: $w_C = \{$ `C2`, `C3`, `C5`, `C6`, `C7`, `C8`, `C9`, `C11`, and `EC`$\}$. In other words, we get 34 candidate pathsets, which includes an ϵ path, a pathset to `EC` and four pathsets (one for each $\delta \leq 4$) to each of the other eight anchor nodes in w_C set: $\{\{\epsilon\} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C2} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C3} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C5} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C6} \ldots\}$.

To keep our following discussion simpler, we restrict our attention to only those pathsets that end at loop heads in C (even though the algorithm considers pathsets to all basic block head and tail nodes and still operates efficiently). The loop heads in C are `C5`, `C7`, and `C2`, and hence the candidate pathsets that we will consider for `(A3-A3)` belong to $\{\{\epsilon\} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C5} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C7} \bigcup \texttt{FPsets}^4_{C5 \rightsquigarrow C2}\}$, for a total of 13 (out of 34) possibilities.

Pruning Based on Paths Taken by Counterexamples

Recall that our path condition based pruning discussed in section 3.3.7 checks that: starting at PCpair `(C5,A3)`, if pathset ξ_A is correlated with pathset ξ_C, then a counterexample at `(C5,A3)` that takes one of the paths in ξ_A must also take one of the paths in ξ_C (to satisfy our correlation criterion in section 3.3.5). Let's assume that one of the counterexamples at `(C5,A3)` that may

have been added either during invariant inference or counterexample propagation is
$\{$ i $\mapsto 0$, sum1 $\mapsto 10$, j $\mapsto 0$, r1 $\mapsto 0$, r2 $\mapsto 0$, r3 $\mapsto 10$, xmm0 $\mapsto 0$, out1 $\mapsto 0$, in1 $\mapsto 200$, in2 $\mapsto 2000$, LEN $\mapsto 12$, $\mathtt{H_C}$[2] $\mapsto (0 \mapsto 10, 200 \mapsto 1, 204 \mapsto 2, 208 \mapsto 3, 212 \mapsto 4, 2000 \mapsto 5, 2004 \mapsto 6, 2008 \mapsto 7, 2012 \mapsto 8, () \mapsto 0)$, $\mathtt{H_A} \mapsto (0 \mapsto 10, 200 \mapsto 1, 204 \mapsto 2, 208 \mapsto 3, 212 \mapsto 4, 2000 \mapsto 5, 2004 \mapsto 6, 2008 \mapsto 7, 2012 \mapsto 8, () \mapsto 0)$, $\ldots\}$.
This counterexample would traverse the path (A3-A3) in program A because it satisfies the corresponding path condition, (r2$+4 \neq$ LEN). However, of the 13 candidate pathsets in C, only 5 would be taken by this counterexample (those that end at C5 at different unrollings and ϵ path). Thus, path condition pruning reduces the candidate correlations from 13 to 5 based on just a single counterexample in this case.

Counterexample-Guided Ranking

After path condition pruning, we are left with five candidate correlations for (A3-A3), namely ϵ, (C5-C5), (C5-C5)^2, (C5-C5)^3, and (C5-C5)^4. Thus, we create five different product-CFGs, each with a *different* newly added product-CFG edge $\omega[(\mathtt{C5}, \mathtt{A3}) \rightarrow (\mathtt{C5}, \mathtt{A3})]$ (the difference is in the correlated pathset) and for each of these newly created product-CFGs, we propagate the counterexamples at (C5,A3) across ω, adding more counterexamples to (C5,A3). All the five product-CFGs are then added to the backtracking search tree. Notice that the same set of counterexamples when added to the product-CFG node (C5,A3) after propagation would be different for each of the product-CFGs because they would have traversed different program paths in C while being propagated across the new product-CFG edge $\omega[(\mathtt{C5}, \mathtt{A3}) \rightarrow (\mathtt{C5}, \mathtt{A3})]$.

As an instance, for the product-CFG that correlates (A3-A3) with (C5-C5)^2, the counterexample used above for discussing the pruning based on paths taken by counterexample, would yield a new propagated counterexample at (C5,A3) obtained by simply propagating it once over the newly added product-CFG edge $\omega[((\mathtt{A3} - \mathtt{A3}), (\mathtt{C5} - \mathtt{C5})^2])$:
$\{$ sum1 $\mapsto 27$, j $\mapsto 2$, r2 $\mapsto 4$, r3 $\mapsto 10$, xmm0 $\mapsto$ 0x20000000150000000C00000005, $\ldots\}$.

[2]The concrete state for memory array is represented using mappings of the form (addr $\mapsto$ data), which implies that the value (byte) stored in memory at address "addr" is "data" and (() $\mapsto$ data) represents the default value "data" for the remaining address space.

Similarly, for the product-CFG that correlates `(A3-A3)` with `(C5-C5)`^{4}, the counterexample after propagating once would be:
{ `sum1` $\mapsto$ 80, `j` $\mapsto$ 4, `r2` $\mapsto$ 4, `r3` $\mapsto$ 10, `xmm0` $\mapsto$ 0x20000000150000000C00000005, ...}.

After propagation, we compute the strongest invariant cover for the updated set of counterexamples at `(C5,A3)` for each of the five candidate product-CFGs. Now, we claim that for the correct correlation, the updated invariant cover at `(C5,A3)` would likely relate more variables in A to variables in C through affine relations. Conversely, for incorrect correlations, the invariant cover at `(C5,A3)` would likely relate fewer variables in A. To see this more concretely, consider the live variables $\texttt{xmm0}_0$, $\texttt{xmm0}_1$, $\texttt{xmm0}_2$, and $\texttt{xmm0}_3$ in A, where $\texttt{xmm0}_i$ is shorthand for `xmm0[(32*`i`+31):(32*`i`)]`. For the incorrect correlations, only some of these four variables may get related by the invariant cover at `(C5,A3)` — e.g., for the product-CFG that correlates `(A3-A3)` with `(C5-C5)`^{2}, the variables $\texttt{xmm0}_0$ and $\texttt{xmm0}_1$ would be related to `sum1` through `(sum1 =` $\texttt{xmm0}_0$ `+` $\texttt{xmm0}_1$ `+ r3)`, but $\texttt{xmm0}_2$ and $\texttt{xmm0}_3$ would not get related to any variables in C. However, for the product-CFG that correlates `(A3-A3)` with `(C5-C5)`^{4}, all four parts of `xmm0` would get related to `sum1` through the invariant cover `(sum1 =` $\texttt{xmm0}_0$ `+` $\texttt{xmm0}_1$ `+` $\texttt{xmm0}_2$ `+` $\texttt{xmm0}_3$ `+ r3)`. For this reason, this latter product-CFG would be ranked higher than the other four product-CFGs in our algorithm.

Thus, ranking helps the algorithm in arriving at the correct correlation in the first attempt during the best-first search procedure in this example.

Pruning Based on Memory Relations

To demonstrate our pruning based on memory relations, consider a partial product-CFG that has almost all the required correlations, as shown in fig. 3.3c, except that it has not yet correlated the pathset `(A10-A2)` starting at PCpair `(C7,A10)`. Among the various correlations for ξ_A = `(A10-A2)`, two candidates pathsets ξ_C in C are `(C7-C2)` and `(C7-C2-C5-C7-C2)`. However we see that while the first candidate updates the memory only once (through writes to `out1[i]` and `out2[i]`), the second candidate updates the memory twice. Thus, the latter candidate correlation `(C7-C2-C5-C7-C2)` is likely to generate an invariant cover (for the propagated counterexamples)

that does not relate the two memory states $\mathtt{H_C}$ and $\mathtt{H_A}$, causing it to be pruned out.

In general, such pruning is very effective in the presence of writes to the memory where the addresses of the writes cannot be characterized at compile time. This pruning strategy is somewhat similar, albeit more flexible and general, to the correlation strategy used in Necula's translation validator [52] where the algorithm required one-to-one correspondence between accesses to the memory for correlation.

3.3.9 Contrast with SPA Algorithm

SPA algorithm [11] is one of the closest competing correlation algorithm to *Counter* in terms of its capabilities. Both SPA and the proposed *Counter* algorithm are data-driven because they rely on data (or counterexamples) to predict (or prioritize) the correlations through path correlations or relations on machine state values. However, *Counter* represents a significant generalization and improvement over the SPA approach because:

- *Counter* does not restrict the correlation condition to be one of the enumerated alignment predicates. Instead it takes a more flexible approach where it simply uses the number of relations in the strongest invariant cover for the concrete counterexamples known so far. This flexibility in *Counter* eliminates the dependence on the availability of the required alignment predicate (which can be quite complex as shown for the example program-pair in fig. 3.3).
- Instead of proposing a single product-CFG (or PAA), *Counter* formulates the algorithm as a best-first search strategy to avoid getting stuck inside a local search subspace. However, our experimental evaluation discussed in (section 5.2) demonstrate that the algorithm converges to the required product-CFG in the first attempt with a very high probability.
- Correlation of pathsets (instead of individual paths) avoids the explosion in the number of required correlations. Our experiments confirm that our method of using *(μ, δ)-unrolled pathsets* (section 3.3.4) along with our correlation criterion (section 3.3.5) does not compromise robustness.

Chapter 4

Counterexample-Guided Invariant Inference

We introduced the notion of the *inductive node-invariants* i.e. invariants associated with the nodes of a given CFG in section 2.2.2. In the context of product-CFG, the *node-invariants* represent the relations between the state-elements of the program-pair and the node-invariant at the start node of the product-CFG is initialized by equating the program arguments and non-temporary memory states of the input program-pair. The inductive invariants at other nodes of the product-CFG are identified through a sound and over-approximate inference procedure. Given a correlation algorithm, the robustness and applicability of an equivalence checker is largely determined by the capabilities of this invariant generation procedure. If the inferred node-invariants are strong enough to prove equivalent observable behavior, then the product-CFG along with the inferred invariants represent the witness (or proof) of equivalence.

As an instance, the first row in fig. 2.4f shows the node-invariant at the start node `(C0,A0)` of the product-CFG shown in fig. 2.4e. The second and third rows in fig. 2.4f (corresponding to product-CFG nodes `(C2,A2)` and `(C4,A4)` respectively) show the invariants inferred using an invariant generation procedure and are strong enough to inductively prove the equivalence of observables at exit represented using the invariant at exit-node `(EC,EA)`. The observables at exit

in this case include the return values `sum` and `r3`, and the non-temporary memory state denoted using $\mathtt{H_C}$ and $\mathtt{H_A}$ for specification program C and the implementation program A respectively.

The problem of automatically identifying these inductive node-invariants that result in an equivalence proof for (a series of) complex compiler transformations is challenging and is undecidable in general. Further, as noted in observation-B in section 3.1, the equivalence checkers usually restrict the choices for correlated PCs that constitute the product-CFG nodes to achieve tractability. If the "ideal" product-CFG that yields a provable bisimulation requires a correlated PC outside these choices, the equivalence checker relies on the invariant inference procedure to efficiently generate expressive invariants to bridge this gap between the ideal correlation and a correlation that only consider the restricted choices for correlated PCs.

For an important class of equivalence checkers that are based on incremental correlation (like [14, 28]), invariant inference procedure is called multiple times at each incremental step for the partial product-CFG constructed so far and the inductive invariants inferred for the partial product-CFG are used to guide the search for future correlations. For instance, the incremental correlation algorithm *Counter* presented in this thesis in section 3.3 uses the invariants inferred for the partial product-CFG for guiding the *best-first search procedure* at each step through pruning and ranking strategies.

In this section, we present a static invariant inference algorithm that is both powerful enough to find inductive invariants between state elements of programs that have large syntactic gap across them and is efficient enough to be used with incremental correlation algorithms. We first discuss the prior work on invariant inference techniques in section 4.1 and then present the details of our proposed invariant inference algorithm in section 4.2.

4.1 Prior work on Invariant Inference Techniques

The broad area of formal verification can be classified into two different verification problems: (1) Safety verification [16, 12, 7, 30] where the goal is to verify that the given program adheres to the specified assertions, (2) Relational verification [63, 11, 14, 40, 25, 26, 17, 49] which attempts

to verify a given property across a pair of programs or different runs of the same program. The literature on invariant inference techniques used in formal verification is vast and can also be broadly categorized into techniques used for invariant inference for safety verification and techniques used for invariant inference for relational verification. Since equivalence checking involves verifying equivalent observable behavior across two input programs, it belongs to the relational verification domain. We thus briefly discuss here the state-of-the-art invariant inference techniques used for relational verification.

4.1.1 Program Execution Based Techniques

Program execution-based techniques involve executing the two programs on real inputs and using linear-algebraic techniques on the observed values to guess invariants [63, 41, 23, 11, 53]. These techniques although powerful, require high-coverage execution traces. However, this requirement of access to execution traces limits the potential applications of equivalence checking. For example, an important application of equivalence checking is translation validation (e.g., within a compiler): unless the programmer is exercising profile-guided optimizations, it is rare for the compiler (or the validator) to have access to real (user-provided) high-coverage testcases. Similarly, for superoptimization and synthesis purposes [3, 60, 4, 46, 55, 62], the search algorithm usually employs random testcases and access to user-provided high-coverage testcases is seldom assumed. In absence of high-coverage execution traces, the invariants inferred using data-driven techniques can be imprecise and their completeness depends on the coverage of the execution traces.

Recent work on data-driven program alignment for equivalence checking [11] suggests that these testcases may be generated randomly or through bounded model-checking. However, this seems impractical for most cases. Figure 4.1 shows an example program to demonstrate that random testcases would usually provide very low coverage (insufficient for use by the verifier), and bounded model-checking has severe scalability issues.

The input to the program in fig. 4.1 includes the state of the non-temporary memory (where the link list is stored). The non-temporary memory includes the heap memory and the memory for

```
C0: void linkListCount(node *head) {
C1:   int count = 0;
C2:   for (; head; head = head->next) { count++ }
C3:   if (count > 1000) {
C4:     ...
C5:   }
C6: }
```

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

the global variables. In this example program, the identification of an input memory-state that would allow the program to reach program location `C4` seems hard. It is highly improbable for random memory-states to provide the desired coverage, and model-checking algorithms would face severe scalability challenges in trying to identify inputs with the desired coverage for this example. In general, identifying high-coverage tests for any program which has multiple loops and/or memory dependencies across loops/loop-iterations is usually hard (the general problem is undecidable).

Sharma et al. [61] tried to address this shortcoming of the program execution-based techniques using a "guess-and-check" algorithm that starts by guessing an invariant from a specified grammar using the concrete values observed in the execution traces. In order to achieve precision, it refines the invariant guess using SMT solver generated counterexamples for not-provable queries during the checking phase. One of the shortcomings of prior work by Sharma et al. [61] is that it does not handle the bitvector arithmetic for invariant inference between the program variables and models the variables as integers; such representation using integers may result in imprecise UB assumptions (based on overflow/wrap-around semantics) that may not be able to prove a transformation correct. Representing the program variables using bitvectors lends robustness in the translation validation context.

A pure data-driven invariant inference procedure (that only uses the program execution traces for actual inputs for inference) based on bitvector representation of the state-elements can potentially infer precise invariants using integer arithmetic because developers usually do not write programs that rely on wrap-around semantics of the program variables. The concrete values generated

using program execution on real inputs, thus, do not exhibit wrap-around behaviors. But the concrete models generated using SMT solver queries (aka counterexamples) may not be restricted to avoid the wrap-around semantics and integer arithmetic based invariant inference would result in weaker (or imprecise) invariants in this case.

For example, consider the program-pair shown in fig. 3.3. One of the required inductive invariant at product-CFG node `(C3,A6)` to prove equivalence is `(r2 + r3 - sum = 0)`. The execution traces observe the values {r2 ↤ 0, r3 ↤ 4, sum ↤ 4} and {r2 ↤ 4, r3 ↤ 0, sum ↤ 4}. Here, `r2`, `r3` and `sum` are 32-bit variables. The integer arithmetic based invariant inference algorithm using these two concrete assignments would result in {`(r2 + r3 = 4); (sum = 4)`} invariant, which is not inductive and shows the imprecision of the program execution-based invariant inference. One of the counterexample returned by the SMT solver for queries to check if the inferred invariant (i.e. {`(r2 + r3 = 4); (sum = 4)`}) is inductively provable is {r2 ↤ 2147483649, r3 ↤ 2147483649, sum ↤ 2}. When this SMT solver generated counterexample is given to an invariant inference procedure that is based on integer arithmetic and not bitvectors, it would fail to infer the required invariant `(r2 + r3 - sum = 0)` and hence the equivalence checker will not be able to generate the required proof.

Our proposed invariant inference algorithm uses a high-level approach similar to the "guess-and-check" algorithm [61], except we only use the SMT solver's satisfying examples as "data" and do not rely on execution traces. Further, the invariant inference algorithm uses bitvector arithmetic in order to generate precise invariants using the SMT solver generated counterexamples.

4.1.2 Syntax/Enumeration Based Techniques

Syntax/Enumeration based techniques [25, 14, 40, 27] generate invariant guesses based on a grammar, followed by a fixed-point checking phase in which these guessed invariants are iteratively eliminated until only the inductively-provable invariants remain. In order to achieve scalability, the guessing grammars used with these techniques are finite and tractable, typically involving equality or inequality relations between two program variables. Invariants generated using these simple equality and inequality relations based grammars do not suffice for programs that have

large syntactic gap across them. The required invariants across the source program written in high-level language like C and the implementation program in assembly are usually of affine shape i.e. ($\sum_{i=1}^{n} c_i.v_i + c_0 = 0$), where v_i represents a state-element in either the specification or implementation program and c_i represents an arbitrary constant. This happens due to various types of arithmetic simplifications and value-to-register mappings performed by an optimizing compiler (e.g. during vectorization). Inferring such invariants using enumeration based techniques is not tractable. In contrast, our proposed counterexample-guided invariant inference algorithm can be used to infer affine-invariants in an efficient manner.

Recurrence based approaches like [33, 37] can infer non-linear invariants, but they work for simplified affine abstractions of programs. Optimized code generated through compilers would usually contain several logical, shift, branching, bit-manipulation, load-store, function-call, etc. opcodes, none of which would fit in their framework of affine programs, and they would usually report equivalence failures for such programs.

4.1.3 Constraint-Solving Based Techniques

Prior work on relational verification [26, 17, 49, 74] has also attempted to reduce the relational verification problem (including the equivalence checking problem) to a safety verification problem by first composing the two given programs into a single program and then using the constraint solving based safety verification techniques on the composed program. In other words, in order to prove that a relation (or equivalence) holds between two given programs P and Q, these techniques show that a pair of pre-condition (Φ) and post-condition (Ψ) holds for the composed program $\{\Phi\}P \sim Q\{\Psi\}$. This hoare-triple formula is encoded a set of *constrained Horn clauses (CHCs)* in a chosen constraint theory. CHCs represent the state-of-the-art logical formalism used for safety verification problems and the satisfiability of the verification conditions is encoded as a set of CHCs (which in this case also guarantees that the relational property holds) can be checked by using off-the-shelf CHC solvers, such as Eldarica [32], Z3 [18] or Data-driven CHC solver [76].

The two main challenges in this constraint solving based relational verification approach are:

1. The techniques required to compose the two programs in order to reduce the relational verification problem to the safety verification problem are non-trivial. Prior-work has either used a manual reduction strategy [5, 26] or have used domain-specific heuristics [74, 17] for obtaining a suitable reduction. Further, given the complexity of the reduction problem, reinforcement-learning based techniques [9] have also been proposed for the same. Using all these techniques, the prior work is only able to handle *structure-preserving* transformations and none of these techniques have been demonstrated across program-pairs that have large structural differences across them. In contrast, our proposed invariant inference algorithm can be used along with a correlation algorithm (like *Counter*) to prove equivalence across transformations like loop unswitching, loop unrolling, loop peeling, loop splitting, etc., that can result in programs with significant structural differences.

2. The constraint-solving based verification involves summarizing the entire execution of the composed program using constrained Horn clauses such that the loop invariants and function summaries for the composed program are encoded as unknown predicates. An off-the-shelf CHC-solver [32, 76] is used to solve the generated set of CHCs and identify loop invariants and relational predicates between the two programs to establish equivalence. *Counterexample-guided abstraction refinement (CEGAR)* based advanced algorithms like IC3 [8], PDR [22] and Horn-ICE [24] have been proposed for these CHC solvers to solve the system of CHCs generated by the verification problems. But, as shown empirically in their evaluation, even after using these advanced algorithms, these techniques work only for small programs which have small syntactic gap across them. In contrast, our experiments in section 5.2 compare programs that are syntactically significantly different because of composition of multiple compiler transformations and can scale for large programs with multiple loops/loop-nesting. The key difference in our approach is: we do not summarize the entire execution in one go and try to infer the invariants incrementally at each product-CFG node and use these inferred invariants to guide future correlation and inference.

It is interesting to note that our algorithm is a dual to iterative refinement based CEGAR techniques used by CHC-solvers. CEGAR based techniques usually start with a high level abstraction (or over-approximation) and perform iterative abstraction refinement until a property (or assertion) is validated or falsified; whereas our algorithm starts with an under-approximation of

program behaviors and adds more behaviors based on counterexamples. CEGAR based invariant inference techniques do not have strict termination guarantees; using such techniques with incremental correlation algorithms (that can handle program-pairs with large structural differences) is not straightforward and has not been demonstrated yet. Our proposed invariant inference algorithm, on the other hand, is incremental and has bounded runtime guarantees for finite invariant grammars and hence can be used to construct a robust translation validator along with an incremental correlation algorithm.

4.2 *Sifer* Algorithm

We identify the following improvement opportunities with respect to the state-of-the-art invariant inference algorithms:

1. The algorithm should work in a static equivalence checking setting i.e. it should not require any input execution traces.
2. The algorithm should be powerful enough to find inductive invariants between state elements of programs that have significant syntactic gap across them and hence do not rely on syntax-based heuristics for inference.
3. The algorithm should be scalable and should have a bounded runtime, so that it can be used with incremental correlation algorithms in an equivalence checker. The invariant inference procedure needs to be efficient enough to be invoked multiple times as a part of an incremental correlation algorithm.

In this section, we present a *counterexample-guided invariant inference algorithm, called* ***Sifer*** [1] to address the limitations of the prior work. The *Sifer* algorithm uses the following key ideas:

1. It is based on a data-flow analysis (DFA) framework. For incrementally constructed graphs (such as the product-CFGs constructed by *Counter* correlation algorithm), the use of DFA formulation allows incremental and efficient invariant inference. At each incremental step,

[1] The name *Sifer* represents a **S**tatic, **S**calable, and **S**imple **I**nvariant in**FER**ence algorithm.

when a new product-CFG edge is added to the partial product-CFG, the invariant inference computation at all nodes does not start from scratch (or the strongest possible invariant, namely `false`) but is refined using the DFA transfer function and meet operators from the invariants inferred at the previous step.

2. It is a static invariant inference algorithm which results in high applicability in an equivalence checking context. Further, such an algorithm has higher scope for automation and practical implementation as it does not depend on availability of high-coverage execution traces for precise invariant inference.
3. Despite absence of execution traces, the algorithm can efficiently infer expressive invariants like affine invariants by using the *counterexamples* (aka concrete models) generated by the SMT solver. In contrast, most of the prior static invariant inference techniques used in equivalence checking were limited to enumeration of predicates drawn from simple grammars based on equality or inequality relations.
4. It also maintains the set of counterexamples (generated or reached after propagation) at each (product-CFG) node. This set of counterexamples are used by the algorithm itself to infer complex invariants in a tractable manner and can be used by other procedures like the correlation algorithm simultaneously.
5. It models the state-elements of the program-pair using bitvectors instead of integers for invariant inference. This results in precise invariants which can prove equivalence for more program-transformation pairs.

4.2.1 Strongest Inductive Invariant Cover

Given a (partial) product-CFG, the *Sifer* algorithm computes the *strongest inductive invariant cover* (Inv_n) at each node n of the product-CFG, where the invariant `Inv` is formed by conjuncting atomic predicates drawn from a grammar $\mathbb{G}$. The *Sifer* algorithm restricts the potential grammars $\mathbb{G}$ such that the candidate invariants `Inv` formed by conjuncting atomic predicates drawn from $\mathbb{G}$ form a semi-lattice.

In the equivalence checking context, these invariants `Inv` associated with the product-CFG nodes

```
int a[20][10];
C0: void foo() {
C1:   for (int i=0; i<20; i++){
C2:     for (int j=0; j<10; j++){
C3:       a[i][j] = i + j;
C4:     }
C5:   }
C6: }
```

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

are mostly used to prove equivalent observable behavior across the specification program C and the implementation program A. Thus, the atomic predicates in these invariants should relate all variables/state-elements in the implementation program A that affect the observable behavior of that program with some variables/state-elements of the specification program C. Although the problem of identifying the required invariants to prove equivalence across any given program-pair is undecidable, invariants formed by conjuncting these atomic predicates relating the state-elements usually suffice for the transformations performed by the modern optimizing compilers.

Since the invariant `Inv` inferred at a product-CFG node n by the the *Sifer* algorithm represents the *strongest inductive invariant cover* within the constraints of the chosen grammar $\mathbb{G}$, in this section, we formally introduce the notion of a *strongest inductive invariant cover*.

Invariant Cover: The node-invariant $\texttt{Inv}_\texttt{n}$ associated with the (product-)CFG node n is also termed as an invariant cover because it is an over-approximate representation of the set of all concrete machine states that are possible at a node n in the abstract domain represented by the grammar $\mathbb{G}$.

For example, consider the program shown in fig. 4.2 and the grammar $\mathbb{G}$ as $\{v \leq c\}$, where v is a program variable and c is an integer constant. Then the invariant $\{i \leq 100 \wedge j \leq 200\}$ is an invariant cover at node `C2` because the invariant is an over-approximation of the set of all possible concrete values of variable i and j at node `C2`.

Henceforth, we will use the term *invariant cover* and *invariant* interchangeably.

Inductive Invariant Cover: An invariant-cover $\texttt{Inv}_{\texttt{n}^\texttt{d}}$ at the (product-)CFG node n^d is inductive across an edge $\omega[n \rightarrow n^d]$ in the CFG, if the following relation holds:

$$\{(\sigma_\omega \wedge \texttt{Inv}_\texttt{n} \wedge \texttt{econd}_\omega) \Rightarrow \texttt{WP}_\omega(\texttt{Inv}_{\texttt{n}^\texttt{d}})\}$$

This is equivalent to:

$$\{(\texttt{SP}_\omega(\texttt{Inv}_\texttt{n}) \Rightarrow \texttt{Inv}_{\texttt{n}^\texttt{d}})\}$$

Here, σ_ω denotes the UB condition and $\texttt{econd}_\omega$ denotes the edge condition for the edge ω. Also, $\texttt{WP}_\omega$(p) represents the weakest precondition of predicate p and $\texttt{SP}_\omega$(p) represents the strongest postcondition of predicate p across the transition through edge ω.

The above condition for inductive invariant cover states that the set of concrete states represented by the invariant cover $\texttt{Inv}_{\texttt{n}^\texttt{d}}$ is a superset of the set of concrete machine states reachable at node n^d through node n. For the example program shown in fig. 4.2 and the invariant cover $\{i \leq 20\}$ at node C1, the invariant cover $\{i \leq 100 \wedge j \leq 200\}$ is an inductive invariant cover at node C2.

Strongest Invariant Cover: The invariant cover Inv is the strongest invariant cover for a given set of concrete machine states Γ (i.e. $\texttt{SInvCover}(\Gamma)$) if the following relation holds:

$$\texttt{Inv} = \texttt{SInvCover}(\Gamma)\ \texttt{if}\ \nexists \texttt{Inv}' \mid (\texttt{Inv} \nRightarrow \texttt{Inv}' \wedge \texttt{Inv} \models \Gamma \wedge \texttt{Inv}' \models \Gamma \wedge \texttt{Inv}' \Rightarrow \texttt{Inv})$$

where $\texttt{Inv} \models \Gamma$ represents that all concrete states in the set Γ satisfy the invariant Inv.

The above condition states that there does not exist another invariant cover Inv′ drawn from the grammar $\mathbb{G}$ such that it is both strictly stronger than the invariant cover Inv and satisfies all concrete models in the set Γ. In other words, it is not possible for an invariant cover Inv′ that the set of concrete states represented by it is a superset of the set Γ and is a strict subset of the set of concrete machine states represented by the invariant cover Inv.

For example, if the set $\Gamma = \{(i \leftarrow 0); (i \leftarrow 5); (i \leftarrow 17)\}$ and the grammar $\mathbb{G} = \{v \leq c\}$, the invariant cover $\{i \leq 20\}$ is not the strongest invariant cover for the set Γ because there exists an invariant cover $\texttt{Inv}' = \{i \leq 19\}$. The strongest invariant cover for the set Γ is $\{i \leq 17\}$.

Strongest Inductive Invariant Cover: The invariant cover $\texttt{Inv}_{\texttt{n}^\texttt{d}}$ is the strongest inductive invariant cover if $\texttt{Inv}_{\texttt{n}^\texttt{d}}$ is both an inductive invariant cover and is the strongest invariant cover of all possible concrete machine states at the node n^d.

For the example program shown in fig. 4.2 and the invariant cover $\{i \leq 20\}$ at node `C1`, the invariant cover $\{i \leq 100 \wedge j \leq 200\}$ is not the strongest inductive invariant cover at node `C2` because there exists an inductive invariant cover $\texttt{Inv}'_{\texttt{n}^\texttt{d}} = \{i \leq 50 \wedge j \leq 100\}$. The strongest inductive invariant cover at node `C2` is $\{i \leq 19 \wedge j \leq 10\}$.

4.2.2 Data-Flow Analysis Framework

Sifer algorithm is based on a forward Data-Flow Analysis (DFA) [2] and uses the Kildall's worklist algorithm to compute the maximum fixed-point (MFP) of the data-flow formulation. The worklist algorithm is an optimized algorithm as compared to the naive iterative algorithm for solving the data-flow formulation for a given CFG. It is based on the premise that input data-flow value at a CFG node is directly determined by the output values at its predecessor nodes and will remain same if the output value of any of the predecessors has not changed. Therefore, instead of re-computing the values for all nodes at each iterative step, the Kildall's worklist algorithm maintains a list of nodes to be processed as a worklist. The algorithm initializes the worklist with the start nodes (or exit nodes in case of a backward data-flow analysis). In each iteration, a node is removed from the worklist and the output value is computed for that node. If the newly computed output value is different from the previous output value for that node, its successors are added to the worklist. For efficiency, a node should not be present in the worklist more than once. The DFA formulation for the *Sifer* algorithm is described in table 4.1.

Domain of DFA Values

The values computed through the DFA are represented by a tuple $(\texttt{Inv}_n, \Gamma_n)$ where $\texttt{Inv}_n$ denotes the invariant at node n and Γ_n denotes a set of counterexamples at node n. As noted in section 3.3.1, a counterexample refers to a concrete machine state in the product-CFG formed by assigning concrete values to the state elements of the abstract machine state for the product-CFG.

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

Domain	$\left\{(\texttt{Inv}_n, \Gamma_n) \;\middle|\; \begin{array}{l}\texttt{Inv}_n \text{ is a conjunction of predicates drawn} \\ \text{from } \mathbb{G} \text{ and } \Gamma_n \text{ is a set of counterexamples}\end{array}\right\}$
Direction	Forward
Boundary condition	$\texttt{out}[n^{start}] = (\texttt{Pre}, \{\})$ (Pre represents the precondition at start node)
Initialization to $\top$	$\texttt{in}[n] = (\texttt{False}, \{\})$ for all non-start nodes
Transfer function	f_ω as specified in fig. 4.3
Meet operator $\otimes$ $(\texttt{Inv}_n, \Gamma_n) \leftarrow (\texttt{Inv}_n^1, \Gamma_n^1) \otimes (\texttt{Inv}_n^2, \Gamma_n^2)$	$\Gamma_n \leftarrow \Gamma_n^1 \cup \Gamma_n^2$, $\texttt{Inv}_n \leftarrow \mathit{SInvCover}(\Gamma_n)$

We refer to these concrete machine states as counterexamples because these are created from the *models* generated by SMT solver for not-provable queries such as those made during invariant inference shown at line number 4 in fig. 4.3.

The *Sifer* algorithm is parameterized for an input grammar $\mathbb{G}$ such that the invariant $\texttt{Inv}_n$ is formed by conjuncting atomic predicates drawn from $\mathbb{G}$. The atomic predicates relate the variables/state-elements of the specification program C and the implementation program A. The *Sifer* algorithm restricts the potential grammars $\mathbb{G}$ to be used such that the candidate invariants enumerated from $\mathbb{G}$ form a semi-lattice with a finite height. The proposed DFA converges in a bounded runtime for such finite grammars making it amenable for incremental correlation based equivalence checkers.

At any node n (other than the start node), the invariant $\texttt{Inv}_n$ represents the **SInvCover** of the set of counterexamples Γ_n at that node, where the $\texttt{SInvCover}(\Gamma_n)$ is defined as the **strongest invariant** generated by conjuncting atomic predicates drawn from the input grammar $\mathbb{G}$ such that all concrete models in set Γ_n satisfy that invariant. The background on the strongest invariant cover is presented in section 4.2.1. Since the possible invariants $\texttt{Inv}_n$ at a node n form a semi-lattice and an invariant $\texttt{Inv}_n$ and the set of counterexamples Γ_n are related by the SInvCover relation, the candidate set of counterexamples at a node also form a semi-lattice and are related by a partial order operator $\leq$ defined as: $(\Gamma^1 \leq \Gamma^2) \Leftrightarrow (\texttt{SInvCover}(\Gamma^2) \Rightarrow \texttt{SInvCover}(\Gamma^1))$. In

other words, Γ^1 is above Γ^2 in the lattice if the `SInvCover` of Γ^1 is stronger than the `SInvCover` of Γ^2. The top ($\top$) value of Γ in this semi-lattice defined by partial order operator ($\leq$) is an empty set because the `SInvCover` of an empty set of counterexamples is the strongest possible invariant i.e. `False`. The bottom value of Γ in this semi-lattice depends on the input grammar $\mathbb{G}$ such that the `SInvCover` for that Γ is the weakest possible invariant cover i.e. `True`.

Initialization of DFA Values

The boundary condition initializes the invariant $\texttt{Inv}_{n^{start}}$ at the start node (for example the node `(C0,A0)` in the product-CFG shown in fig. 3.1c) to the precondition that asserts the equality of non-temporary memory states (`H`) and input arguments while considering ABI and calling conventions. The boundary condition also initializes the counterexample set $\Gamma_{n^{start}}$ to the empty set. For each node n other than the start node, the tuple ($\texttt{Inv}_n$, Γ_n) is initialized to the top-most values in their respective semi-lattices. Thus, the invariant $\texttt{Inv}_n$ is initialized to `False` (i.e. the strongest possible invariant cover) and the counterexample set Γ_n is initialized to an empty set (as the `SInvCover` of empty set is the strongest possible invariant `False`).

Transfer Function and Meet Operator

The evaluation of the transfer function f_ω for an edge $\omega[n \rightarrow n^d]$ in the CFG involves a fixed-point procedure as shown in fig. 4.3. It takes as input the invariant $\texttt{Inv}_n$ and counterexample set Γ_n at node n. The transfer function f_ω also involves the application of the meet operator $\otimes$ with the old DFA value $(\texttt{Inv}_{n^d}, \Gamma_{n^d})$ at the target node n^d of the edge. The output of the transfer function is the new DFA value of the tuple $(\texttt{Inv}_{n^d}, \Gamma_{n^d}) = f_\omega(\texttt{Inv}_n, \Gamma_n, \texttt{Inv}_{n^d}, \Gamma_{n^d})$ at the target node n^d.

At a high-level, the DFA transfer function across an edge $\omega[n \rightarrow n^d]$ from node n to node n^d involves: (1) Identifying the strongest invariant $\texttt{Inv}_{n^d}$ that is weaker than the strongest-postcondition of the invariant at n, $\texttt{Inv}_n$ across edge ω (inductive invariant), and (2) Adding counterexamples to the set Γ_{n^d} based on the proof obligations generated during invariant inference such that their strongest invariant cover is $\texttt{Inv}_{\texttt{n}^\texttt{d}}$.

The DFA's meet operator $\otimes$ at the node n^d involves computing the union of the input counterexample sets ($\Gamma^1_{n^d} \cup \Gamma^2_{n^d}$) and then updating the strongest invariant cover $\texttt{Inv}^{out}_{n^d}$ accordingly.

Algorithm 7: Transfer function for edge $\omega[n \rightarrow n^d]$ along with the meet operator at the target node n^d

1 **Function** $f_\omega(\texttt{Inv}_n, \Gamma_n, \texttt{Inv}_{n^d}, \Gamma_{n^d})$
2 $\quad \Gamma^p_{n^d} \leftarrowtail p_\omega(\Gamma_n)$;
3 $\quad (\texttt{Inv}^{can}_{n^d}, \Gamma^{can}_{n^d}) \leftarrowtail (\Gamma_{n^d} \otimes \Gamma^p_{n^d})$; // Meet operator
4 $\quad$ **while** $\texttt{SAT}(\texttt{Inv}_n \wedge \neg \texttt{WP}_\omega(\texttt{Inv}^{can}_{n^d}), \gamma_n)$ **do**
5 $\quad\quad \gamma_{n^d} \leftarrowtail p_\omega(\gamma_n)$;
6 $\quad\quad (\texttt{Inv}^{can}_{n^d}, \Gamma^{can}_{n^d}) \leftarrowtail (\Gamma^{can}_{n^d} \otimes \{\gamma_{n^d}\})$; // Meet operator
7 $\quad$ **end**
8 $\quad$ **return** $(\texttt{Inv}^{can}_{n^d}, \Gamma^{can}_{n^d})$;
9 **Function** $(\Gamma^1_{n^d} \otimes \Gamma^2_{n^d})$
10 $\quad \Gamma^{out}_{n^d} \leftarrowtail \Gamma^1_{n^d} \cup \Gamma^2_{n^d}$;
11 $\quad \texttt{Inv}^{out}_{n^d} \leftarrowtail \texttt{SInvCover}(\Gamma^{out}_{n^d})$;
12 $\quad$ **return** $(\texttt{Inv}^{out}_{n^d}, \Gamma^{out}_{n^d})$;
13 .

Figure 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.

The fixed-point algorithm implemented by the function f_ω starts by propagating (or executing) the input counterexample set Γ_n across the edge ω using its concrete execution function p_ω. The set of propagated counterexamples at node n^d is denoted by $\Gamma^p_{n^d}$. The propagation of the counterexample set Γ_n across the edge ω to result in the counterexample set $\Gamma^p_{n^d}$ is an optimization step in the algorithm to potentially reduce the number of iterations of the fixed-point loop (line number 4-7 in fig. 4.3). By application of the meet operators $\otimes$, the set of propagated counterexamples $\Gamma^p_{n^d}$ is added to the existing set Γ_{n^d}, and a new candidate set $\Gamma^{can}_{n^d}$ is obtained. The meet operator also returns the new candidate invariant $\texttt{Inv}^{can}_{n^d}$ at node n^d by computing the *strongest invariant cover* of the counterexample set $\Gamma^{can}_{n^d}$.

To check the inductive property of the new candidate invariant $\texttt{Inv}^{can}_{n^d}$, a proof obligation represented through a relational Hoare triple [6, 31] as $\{\texttt{Inv}_n\}\omega\{\texttt{Inv}^{can}_{n^d}\}$ is generated at node n.

This Hoare triple states that if the machine starts at node n such that it satisfies $\{\texttt{Inv}_n\}$, and the edge ω is executed, then the resulting machine state would satisfy $\{\texttt{Inv}_{n^d}^{can}\}$. To discharge proof obligations, the Hoare triple $\{\texttt{Inv}_n\}\omega\{\texttt{Inv}_{n^d}^{can}\}$ is converted (or lowered) to a propositional boolean logic formula at node n of the form $\texttt{Inv}_n \Rightarrow \texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$, where $\texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$ computes the weakest precondition of $\texttt{Inv}_{n^d}^{can}$ across ω. Since in the context of a product-CFG, an edge ω represents a finite pathset of paths with a finite length, we use standard formulations [21, 26] for inferring the weakest-precondition ($\texttt{WP}_\omega$) across the edge ω.

The proof obligation for this Hoare triple boolean formula is discharged through an off-the-shelf SMT solver with quantifier-free bitvector, array and uninterpreted function theories. If the proof succeeds, $\texttt{Inv}_{n^d}^{can}$ holds the strongest-possible inductive invariant at node n^d for the grammar $\mathbb{G}$ and $\Gamma_{n^d}^{can}$ holds the output set of counterexamples at n^d, both of which are then returned by the algorithm.

If the proof obligation for the current candidate invariant $\texttt{Inv}_{n^d}^{can}$ does not succeed, then a counterexample γ_n at node n is returned by the SMT solver. The counterexample γ_n represents a concrete assignment to the program variables or state-elements at the node n such that it satisfies the invariant $\texttt{Inv}_n$ but does not satisfy the weakest-precondition of the candidate invariant across the edge ω, i.e., $\texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$. This new counterexample γ_n is propagated to node n^d by applying the concrete execution function p_ω for the edge $\omega[n \rightarrow n^d]$ and the propagated counterexample γ_{n^d} is added to the counterexample set $\Gamma_{n^d}^{can}$ (line number 5 and 6 in fig. 4.3).

Unlike the counterexample propagation performed at line number 2 of the algorithm (fig. 4.3), the propagation of the newly generated counterexample γ_n across the edge $\omega[n \rightarrow n^d]$ at line number 5 to result in γ_{n^d} is not an optimization but is necessary to weaken the not-provable candidate invariant $\texttt{Inv}_{n^d}^{can}$ by recomputing the strongest invariant cover (`SInvCover`) of the new counterexample set $\Gamma_{n^d}^{can}$. The fixed-point procedure then again discharges the proof obligation for checking the inductive property of this new candidate invariant. The fixed-point procedure exits either when the candidate invariant $\texttt{Inv}_{n^d}^{can}$ is proven inductive or it reaches the bottom of the semi-lattice in which case the computed invariant $\texttt{Inv}_{n^d}^{can}$ is `True`.

4.2.3 Characteristics of the Algorithm

Precise Invariant: *The invariant* $\texttt{Inv}_{n^d}^{can}$ *returned as solution by the data-flow analysis formulation in section 4.2.2 is the* **strongest inductive invariant cover** *at the node* n^d *that can be constructed by conjuncting the atomic predicates drawn from the input grammar* $\mathbb{G}$*.*

The proof of this claim follows from the following properties of the algorithm:

(a) The output of the transfer function f_ω is an *inductive invariant* because when the f_ω computation shown in fig. 4.3 returns, $(\texttt{Inv}_n \wedge \neg \texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can}))$ is `UNSAT` and there is no concrete model that both satisfies $\texttt{Inv}_n$ and does not satisfy $\texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$. In other words, all possible states at node n that satisfy $\texttt{Inv}_n$ also satisfy $\texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$, i.e. $\texttt{Inv}_n \Rightarrow \texttt{WP}_\omega(\texttt{Inv}_{n^d}^{can})$

(b) The inductive claim stated in the above point also implies that when the f_ω computation for an edge $\omega[n \rightarrow n^d]$ exits, the counterexample set Γ_{n^d} is a superset of the concrete states reachable at node n^d from node n, i.e. $\texttt{ConcreteStates}(\texttt{SP}_\omega(\texttt{Inv}_n)) \subseteq \Gamma_{n^d}$.

(c) The invariant $\texttt{Inv}_{n^d}^{can}$ is computed by the application of the `SInvCover` on the counterexample set Γ_{n^d}. As discussed in section 4.2.1, $\texttt{SInvCover}(\Gamma)$ returns the strongest invariant cover `Inv` such that there does not exist another invariant cover $\texttt{Inv}'$ drawn from the grammar $\mathbb{G}$ that is both stronger than the invariant cover `Inv` and satisfies all concrete models in the set Γ. Thus for the edge $\omega[n \rightarrow n^d]$,

$$\nexists \texttt{Inv}'_{\texttt{n}^\texttt{d}} \mid (\texttt{Inv}_{\texttt{n}^\texttt{d}} \not\Rightarrow \texttt{Inv}'_{\texttt{n}^\texttt{d}} \wedge \texttt{Inv}_{\texttt{n}^\texttt{d}} \models \Gamma_{n^d} \wedge \texttt{SP}_\omega(\texttt{Inv}_\texttt{n}) \Rightarrow \texttt{Inv}'_{\texttt{n}^\texttt{d}} \wedge \texttt{Inv}'_{\texttt{n}^\texttt{d}} \Rightarrow \texttt{Inv}_{\texttt{n}^\texttt{d}}).$$

Thus, the invariant $\texttt{Inv}_{n^d}^{can}$ represents both an inductive invariant cover and the strongest invariant cover at the node n^d.

Termination: *If the semi-lattice formed by the candidate invariants drawn from the grammar* $\mathbb{G}$ *has finite height, then the transfer function computation is guaranteed to converge in a bounded number of steps.*

This holds because in each iteration of the fixed-point transfer function algorithm, the new counterexample γ_{n^d} added to the counterexample set Γ_{n^d} does not satisfy the current candi-

date invariant $\texttt{Inv}^{can}_{n^d}$ at n^d. As a result, the next generated invariant candidate $\texttt{new-Inv}^{can}_{n^d} = \texttt{SInvCover}(\gamma_{n^d} \cup \Gamma_{n^d})$ would always be strictly weaker than the previous $\texttt{Inv}^{can}_{n^d} = \texttt{SInvCover}(\Gamma_{n^d})$. In other words, $\texttt{new-Inv}^{can}_{n^d} < \texttt{Inv}^{can}_{n^d}$ always because the $\texttt{new-Inv}^{can}_{n^d}$ includes an extra point γ_{n^d} that was absent in $\texttt{Inv}^{can}_{n^d}$. In the worst case, the number of steps or counterexamples required is upper bounded by the height of the semi-lattice formed by the candidate invariants $\texttt{Inv}^{can}_{n^d}$ because at each step (or by adding each new counterexample) the candidate invariant $\texttt{Inv}^{can}_{n^d}$ gets strictly weaken and at least goes one step down in the semi-lattice till it reaches the weakest possible invariant, i.e. `True`. The height of the semi-lattice formed by the candidate invariants $\texttt{Inv}^{can}_{n^d}$ depends on the grammar $\mathbb{G}$ from which the atomic predicates for the invariant are chosen.

Monotonicity: *The transfer function f_ω across an edge $\omega[n \to n^d]$ is always monotonic. Thus, if the invariant cover $\texttt{Inv}^1_{\texttt{n}}$ given as input to the function f_ω is weaker than the invariant cover $\texttt{Inv}^2_{\texttt{n}}$, then the output $f_\omega(\texttt{Inv}^1_{\texttt{n}})$ is weaker than the output $f_\omega(\texttt{Inv}^2_{\texttt{n}})$.*

The proof of this claim follows from the monotonicity of the strongest postcondition, i.e.,

$$\{\texttt{Inv}^1_{\texttt{n}} \leq \texttt{Inv}^2_{\texttt{n}} \Rightarrow \texttt{SP}_\omega(\texttt{Inv}^1_{\texttt{n}}) \leq \texttt{SP}_\omega(\texttt{Inv}^2_{\texttt{n}})\},$$

and from the *Precise Invariant* claim, i.e. the transfer function output is the strongest inductive invariant cover such that

$$\{\texttt{SP}_\omega(\texttt{Inv}_{\texttt{n}}) \leq f_\omega(\texttt{Inv}_{\texttt{n}})\} \textit{ and } \{\texttt{SP}_\omega(\texttt{Inv}^1_{\texttt{n}}) \leq \texttt{SP}_\omega(\texttt{Inv}^2_{\texttt{n}}) \Rightarrow f_\omega(\texttt{Inv}^1_{\texttt{n}}) \leq f_\omega(\texttt{Inv}^2_{\texttt{n}})\}$$

Both the monotonicity of the strongest postcondition computation and the *Precise Invariant* claim together imply that: $\texttt{Inv}^1_{\texttt{n}} \leq \texttt{Inv}^2_{\texttt{n}} \Rightarrow f_\omega(\texttt{Inv}^1_{\texttt{n}}) \leq f_\omega(\texttt{Inv}^2_{\texttt{n}})$

As an aside, we note that while our data-flow analysis algorithm computes the maximum fixed-point of the semi-lattice, the generated solution may be weaker than the "meet-over-paths" solution. This is true because our transfer functions, though monotonic, are not necessarily distributive.

Algorithm 8: Algorithm to find the most precise inductive $\mathcal{A}$-*invariant*

1 **Function** $\widehat{\texttt{Post}[\tau]}(\texttt{Inv}_n)$
2 $\quad \texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}} \leftarrow\!\shortmid \texttt{False}$;
3 $\quad$ **while** $\texttt{SAT}(\texttt{Inv}_\texttt{n} \wedge \neg \texttt{WP}_\omega(\texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}}), \gamma_n)$ **do**
4 $\quad\quad \gamma_{n^d} \leftarrow\!\shortmid p_\omega(\gamma_n)$;
5 $\quad\quad \texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}} \leftarrow\!\shortmid \texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}} \sqcup \beta(\gamma_{n^d})$;
6 $\quad$ **end**
7 $\quad$ **return** $\texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}}$;

Figure 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.

4.2.4 Comparison with the abstract interpretation based prior work

Abstract interpretation based theoretical framework was proposed by [58, 68] to compute the most precise abstraction of a program in a given abstract domain using an SMT solver. The transfer function algorithm f_ω in our data flow analysis formulation can be considered as an adaption of this theoretical framework, where the grammar $\mathbb{G}$ corresponds to the abstract domain $\mathcal{A}$, the execution function p_ω corresponds to the application of the concrete transformer $\texttt{Post}[\tau]$ and the transfer function f_ω to compute the strongest inductive invariant cover $\texttt{Inv}$ corresponds to the application of the $\widehat{\texttt{Post}[\tau]}$ operator to find the most precise inductive $\mathcal{A}$-*invariant*. However, the proposed transfer function f_ω shown in fig. 4.3 has significant improvements as compared to the theoretical $\widehat{\texttt{Post}[\tau]}$ operator used by the prior work shown in fig. 4.4.

Worst Case Bound: The first major difference is that the $\widehat{\texttt{Post}[\tau]}$ operator always initializes the candidate invariant $\texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}}$ to $\texttt{False}$ and weakens it till it is provably inductive. In contrast, the transfer function f_ω in the proposed DFA framework does not necessarily begin the invariant computation from the topmost element in the semi-lattice (i.e. $\texttt{False}$). It first propagates (or executes) the counterexamples in the set Γ_n over the edge ω using its concrete execution function p_ω and adds the resulting counterexamples to the set $\Gamma^p_{n^d}$ (line number 2 in fig. 4.3). The transfer function f_ω then initializes $\texttt{Inv}^{\texttt{can}}_{\texttt{n}^\texttt{d}}$ with the invariant cover obtained after applying the meet operator on the old DFA value at the node n^d and the concrete set $\Gamma^p_{n^d}$ obtained after

propagation (line number 3 in fig. 4.3).

If "h" denotes the height of the semi-lattice formed by the candidate invariants drawn from the grammar $\mathbb{G}$ and 'N' denotes the number of nodes in a given product-CFG then the total number of computations (or SMT solver queries) required by the *Sifer* algorithm (using f_ω as DFA transfer function) for computing the strongest inductive invariant cover at each node of the given product-CFG is upper-bounded by $O(N * h)$. In contrast, the $\widehat{\texttt{Post}[\tau]}$ operator based invariant inference algorithm would always start from the `False` invariant and has the worst case bound $O(N * h^2)$.

Further, the average number of computations (or the SMT solver calls) required are much less because other than starting from the previous invariant cover DFA value instead of `False`, the proposed f_ω operator also maintains the counterexamples at the (product-CFG) nodes and executes these counterexamples using the concrete execution function p_ω (line number 2 in fig. 4.3) across the edge ω to obtain new concrete mappings at the target node of the edge. These counterexamples reduce the number of iterations or SMT solver queries required till the inductive invariant is reached. Further, the set of counterexamples maintained at each (product-CFG) node by the proposed DFA formulation can also be used by other procedures like the correlation algorithm and brings the static equivalence checkers more close to the data-driven equivalence checkers in terms of available data to guide the search for the proof.

Practical Algorithm: Another important difference is that the $\widehat{\texttt{Post}[\tau]}$ operator does not provide insights on the practical implementation of the representation function β that converts a given concrete assignment to an abstract value and $\sqcup$ operator that joins two given abstract values. Our proposed transfer function, on the other hand, performs *set union* to join two DFA values and uses the `SInvCover` operator explained in section 4.2.1 to find the strongest invariant cover for a set of concrete states. We present the algorithm used to find the `SInvCover` for selected grammars in section 4.2.5.

Incremental Algorithm: For incrementally constructed graphs (such as the product-CFGs constructed by *Counter* algorithm or [14]), the *Sifer* algorithm as compared to the $\widehat{\texttt{Post}[\tau]}$ operator allows incremental yet precise invariant inference.

The incremental correlation algorithms, at each step take a partial product-CFG π as input and construct a new product-CFG π_{new} by adding a new edge $\omega[n \rightarrow n^d]$ to it. Based on the proposed DFA formulation discussed in section 4.2.2, *Sifer* algorithm does not start the invariant computation from scratch after adding a new edge to the partial product-CFG π, but re-uses the DFA values $(\texttt{Inv}, \Gamma)$ computed for π to initialize the DFA values at the nodes of π_{new}. Since, the addition of a new edge can only increase the number of behaviors (or constraints) at the target node n^d of the newly added edge and the reachable nodes from the target node n^d in the product-CFG π_{new}. Thus, the strongest invariant cover at the nodes of the new product-CFG π_{new} after adding the edge could only get weakened, as compared to the invariant cover at the same nodes in the old product-CFG π. The strongest inductive invariant cover $\texttt{Inv}_\texttt{n}\{\pi\}$ at the node n in the partial product-CFG π is stronger than the strongest inductive invariant cover $\texttt{Inv}_\texttt{n}\{\pi_{new}\}$ at the node n in the partial product-CFG π_{new}, i.e. $\forall n \in \pi,\ (\texttt{Inv}_\texttt{n}\{\pi\} \Rightarrow \texttt{Inv}_\texttt{n}\{\pi_{new}\})$

Starting from the DFA values at the respective nodes of the previous product-CFG π, the data-flow analysis construction ensures that the Maximum Fixed Point (MFP) solution i.e. the strongest inductive invariant cover is obtained at every node of the new product-CFG π_{new}. Further, the nodes that are not reachable from the target node of the newly added edge, the invariant cover in the old product-CFG π is also the strongest invariant cover in the new product-CFG π_{new} and no computation needs to be performed for these nodes. Thus, the data-flow analysis formulation of the invariant inference algorithm while performing incremental computation towards invariant computation remains precise for all (product-CFG) nodes at each step of an incremental correlation algorithm [14, 28].

4.2.5 Computation of the SInvCover()

SInvCover() for Affine Invariant Grammar

The atomic predicates belonging to an affine invariant grammar are of the form

$$\sum_{i=1}^{n} c_i * v_i + c_0 = 0 \tag{4.1}$$

where, v_i represent state-elements of the program (including e.g., values in variables and/or registers/memory locations) and c_i represent constant values satisfying the above equation. In case of an equivalence checker, the program variables v may belong to the specification program C or the implementation program A and an atomic predicate thus represents a relation between variables across the two programs. The precise modeling for programming languages like C and assembly involves using *bitvector representation* for the state-elements instead of a real number or integer representation. In this modeling, for an atomic predicate as specified in eq. (4.1), all program variables v_i and the constants c_i are bitvectors of the same width (say w). For example, the program state-elements v_i and the constants c_i satisfying the affine relation can be bitvectors of width 32. Further, the multiplication ($*$) and addition ($+$) operators in this modeling represent two's complement bitvector-multiplication and bitvector-addition respectively with wrap-around semantics for overflow.

In the geometrical representation, for real numbered values, i.e. $v_i \in \mathbb{R}$ ($i \in [1, n]$), an affine-invariant represents the basis for a *linear subspace* (e.g., point, line, plane, etc.) of $\mathbb{R}^n$ [61]. For bitvector arithmetic, the geometrical interpretation is less clear: e.g., consider two program variables $v_1, v_2 \in \mathbb{B}_{32}$ (where $\mathbb{B}_w$ represents the set of all bitvectors of width w). Also, consider two affine-invariants $\texttt{Inv}_1 = ((v_1 = 2^{31}) \wedge (v_2 = 0))$ and $\texttt{Inv}_2 = ((v_2 = 2.v_1) \wedge (v_2 = 4.v_1))$. While $\texttt{Inv}_1$ represents a *singleton point* $= (2^{31}, 0)$ in $\mathbb{B}_{32}^2$, $\texttt{Inv}_2$ represents an intersection of two "bitvector lines" of slopes 2 and 4 respectively; these two lines intersect on two points, namely $(v_1 = 0 \wedge v_2 = 0) \vee (v_1 = 2^{31} \wedge v_2 = 0)$. However, we will still call the bitvector invariant formed by conjuncting atomic predicates of the form specified in eq. (4.1), *affine-invariant*, even though they may not actually represent a linear subspace for bitvectors.

If matrix $\mathcal{M}$ is formed by using the concrete points in the counterexample set Γ as its rows, then for real number program values, i.e. $v_i \in \mathbb{R}$ ($i \in [1, n]$), the `SInvCover(`Γ`)` involves computing the basis of the column space of the matrix $\mathcal{M}$. For bitvector domain, the algorithm presented by Müller-Olm et al. [50] computes the nullspace basis of the matrix $\mathcal{M}$ having bitvector values and thus can be used for `SInvCover()` computation for bitvector valued counterexample set. Given the matrix $\mathcal{M}$ with n bitvector variables of width w, i.e. $\mathcal{M} \in \mathbb{B}_w^n$, the time taken by the nullspace basis computation algorithm presented by Müller-Olm et al. [50] is $O(n^3 * log(w))$. Further, it is proven by Müller-Olm et al. [50] that the length of a strictly increasing chain of $\mathbb{B}_w^n$-modules,

which corresponds to the height of the semi-lattice formed by the affine bitvector invariants with n number of variables of width w, is bounded by $(n * w + 1)$.

SInvCover() for Binary-Relation Invariant Grammar

The *binary-relation* invariant grammar has atomic predicates of the form $(v_i \ \mathcal{R} \ v_j)$, where, v_i and v_j represent two bitvector state-elements of the program (including e.g., values in variables and/or registers/memory locations) and $\mathcal{R}$ represents a binary relation such as inequality ($\leq$) or equality ($=$). If the total number of interesting program variables for which the binary relation $\mathcal{R}$ needs to be computed is n, then the algorithm to compute `SInvCover(`Γ`)` across these variables for a given counterexample set Γ involves enumerating all n^2 atomic predicates of the form

$$\forall (i,j) \ \mid \ i \neq j \ and \ (i,j) \in [1,n] : \ \{v_i \ \mathcal{R} \ v_j\}$$

All atomic predicates that are not satisfied by the set of counterexamples (Γ) are then eliminated and the conjunction of remaining atomic predicates is returned as the strongest invariant cover by the `SInvCover()` procedure.

4.2.6 Explaining the Sifer algorithm through an example

Consider, one of the affine invariant required to be inferred at node `(C3,A6)` in the product-CFG shown in fig. 3.2c is **(r1 - b - 4*i = 0)**.

Prior static invariant inference techniques would generate the affine invariant guesses of the form $\sum_{i=1}^{n} c_i v_i + c_0 = 0$, where v_i represents a bitvector program variable (including registers in assembly) and c_i represents a bitvector constant. The guessing phase is followed by a fixed-point checking phase which iteratively prunes away the guesses which are not inductively provable. For grammars like affine invariants (as required in this case), the total number of guesses and the required proof queries are intractably large (in the order of 2^w for program variables with a bitwidth of w). For illustration, we discuss the inference of this invariant using the proposed *Sifer*

algorithm which uses the counterexamples (aka concrete states) generated by the SMT solver to tractably infer the required invariant. The `SInvCover()` computation procedure for the required affine invariant grammar is already discussed in section 4.2.5.

Consider the partial product-CFG (π) which only has nodes `(C0,A0)` and `(C3,A2)` and two edges `((C0,A0)` $\rightarrow$ `(C3,A2))` and `((C3,A2)` $\rightarrow$ `(C3,A2))`. Also, consider the strongest invariant cover $\texttt{Inv}_n$ and the counterexample set Γ_n at node n = `(C3,A2)` computed by running *Sifer* on π as

$$\texttt{Inv}_n = \begin{pmatrix} (LEN = 1000); (r1 - i = 0); \\ (r2 - sum = 0); (b = 32); \end{pmatrix}, \Gamma_n = \begin{Bmatrix} LEN & i & r1 & b \\ 1000 & 498 & 498 & 32 \\ 1000 & 499 & 499 & 32 \end{Bmatrix}$$

Here, the counterexample set is shown using the matrix representation and each counterexample or concrete state is the set is shown as a row of the matrix. Also, the mappings for the memory and other program variables are not shown for ease of exposition.

Edge ((C3,A2) $\rightarrow$ (C3,A6)) Added:

As a incremental step, the node `(C3,A6)` along with the edge `((C3,A2)` $\rightarrow$ `(C3,A6))` is added to get the new partial product-CFG (π_{new}). *Sifer* being an incremental invariant inference algorithm, reuses the DFA value computed for the product-CFG π to initialize the values for the product-CFG π_{new} (as discussed in section 4.2.4). Also, the DFA value at the newly added node n^d = `(C3,A6)` is initialized to ($\texttt{Inv}_{\texttt{n}^\texttt{d}} = \texttt{False}$, $\Gamma_{n^d} = \varnothing$).

Since, both the nodes `(C0, A0)` and `(C3,A2)` are not reachable from the target node of the newly added edge `((C3,A2)` $\rightarrow$ `(C3,A6))`, the invariant inference is not re-run for these nodes and only involves computing the transfer function f_ω across the newly added edge $\omega[(\texttt{C3}, \texttt{A2}) \rightarrow (\texttt{C3}, \texttt{A6})]$.

The first step in the f_ω computation is to execute (or propagate) the counterexamples in the set Γ_n over the edge $\omega[(\texttt{C3}, \texttt{A2}) \rightarrow (\texttt{C3}, \texttt{A6})]$. The first counterexample has the concrete mapping for register `r1 = 498` and hence this counterexample does not satisfy the condition for the edge $(\texttt{C3}, \texttt{A2}) \rightarrow (\texttt{C3}, \texttt{A6})$ (i.e. $(\texttt{r1} + 1 == \texttt{LEN}/2) \wedge (\texttt{i} < \texttt{LEN})$). The second counterexample (with concrete mapping for register `r1 = 499`) satisfies the edge condition and is executed successfully on

the edge. Thus, in this case, a singleton set $\Gamma^{p}_{n^d}$ obtained after execution of the counterexamples is: $\Gamma^{p}_{n^d} = \begin{Bmatrix} LEN & i & r1 & b & r3 \\ 1000 & 500 & 2032 & 32 & 4032 \end{Bmatrix}$

The new DFA value $(\texttt{Inv}^{can}_{n^d}, \Gamma^{can}_{n^d})$ is obtained after applying the meet operator to the set $\Gamma^{p}_{n^d}$ which is a singleton set in this case and the previous value of Γ_{n^d} which is empty in this case.

$$\texttt{Inv}^{can}_{n^d} = \begin{pmatrix} (LEN = 1000); (i = 500); (r1 = 2032) \\ (b = 32); (r3 = 4032); \end{pmatrix} \Gamma^{can}_{n^d} = \begin{Bmatrix} LEN & i & r1 & b & r3 \\ 1000 & 500 & 2032 & 32 & 4032 \end{Bmatrix}$$

The algorithm then proceeds by checking if the candidate invariant cover $(\texttt{Inv}^{can}_{n^d})$ is inductively provable by discharging the proof obligation for the weakest precondition of the candidate invariant cover $(\texttt{Inv}^{can}_{n^d})$ across the edge $\omega[(\texttt{C3}, \texttt{A2}) \rightarrow (\texttt{C3}, \texttt{A6})]$. In this case, the proof obligation results `UNSAT` as the above inferred invariant cover from the propagated counterexample set itself is inductively provable across the edge ω and the invariant inference DFA transfer function returns the above obtained counterexample set and the invariant cover as output. This highlights the advantage of the incremental characteristic to the *Sifer* algorithm that minimizes the SMT solver queries by re-using the previous counterexamples through concrete execution.

Edge ((C3,A6) → (C3,A6)) Added:

At the next incremental step, the above partial product-CFG π_{new} is considered as the input partial product-CFG π and a new edge $\omega[(\texttt{C3}, \texttt{A6}) \rightarrow (\texttt{C3}, \texttt{A6})]$ is added to obtain the new product-CFG π_{new}. Being an incremental algorithm, the evaluation of the transfer function f_ω across the newly added edge starts with the DFA value computed at the previous step. Thus the invariant $\texttt{Inv}_n$ and the counterexample set Γ_n at node $n =$ `(C3,A6)` is same as shown above as the return values for the previous step. Also, for the edge $\omega[(\texttt{C3}, \texttt{A6}) \rightarrow (\texttt{C3}, \texttt{A6})]$, the node n and n^d are same, so the initial counterexample set Γ_{n^d} at node n^d = (C3,A6) is same as Γ_n.

The counterexample in the singleton set Γ_n satisfies the condition for the edge $\omega[(\texttt{C3}, \texttt{A6}) \rightarrow (\texttt{C3}, \texttt{A6})]$ (i.e. $(\texttt{r1} + \texttt{16!} = \texttt{r3}) \wedge (\texttt{i} < \texttt{LEN})$) and is executed successfully on the edge. The set $\Gamma^{p}_{n^d}$ obtained after execution of the counterexample set Γ_n is:

$$\Gamma^{p}_{n^d} = \begin{Bmatrix} LEN & i & r1 & b & r3 \\ 1000 & 504 & 2048 & 32 & 4032 \end{Bmatrix}$$

The new DFA value $(\texttt{Inv}_{n^d}^{can}, \Gamma_{n^d}^{can})$ is obtained after applying the meet operator to this above $\Gamma_{n^d}^{p}$ set and the previous value of Γ_{n^d} both of which are singleton sets in this case.

$$\texttt{Inv}_{n^d}^{can} = \begin{pmatrix} (LEN = 1000); (b = 32); (r3 = 4032); \\ (i * 1073741824 = 0); \\ (r1 - b - 4 * i = 0); \end{pmatrix} \Gamma_{n^d}^{can} = \left\{ \begin{matrix} LEN & i & r1 & b & r3 \\ 1000 & 500 & 2032 & 32 & 4032 \\ 1000 & 504 & 2048 & 32 & 4032 \end{matrix} \right\}$$

In this case, the inferred invariant cover $\texttt{Inv}_{n^d}^{can}$ is inductively provable across the edge $\omega[(\texttt{C3}, \texttt{A6}) \rightarrow (\texttt{C3}, \texttt{A6})]$ in the first iteration itself and has the required invariant: **(r1 - b - 4*i = 0)**. But, in general, if the inferred invariants are not inductively provable then a counterexample (γ_n) would be returned by the SMT solver at node n = (C3,A6). The returned counterexample is then executed over the edge $\omega[(\texttt{C3}, \texttt{A6}) \rightarrow (\texttt{C3}, \texttt{A6})]$ and added to the candidate counterexample set. The candidate counterexample set with this new concrete state would then result in a weaker invariant cover candidate. The algorithm performs this inductive check followed by counterexample execution loop until the inferred invariant cover becomes inductively provable.

Chapter 5

Unoptimized-IR-to-Optimized-Assembly Translation Validator

In the context of compilation of imperative programming languages, the equivalence needs to be computed between the high-level program specification and the low-level assembly program implementation. In this manifestation of the equivalence checking problem, there is a large syntactic gap between the specification and the implementation and the required invariants to prove equivalence corresponds to relationships between variable states in high-level program representation and register/memory-location states in low-level hardware syntax. Further, computing these relationships automatically across two different syntaxes in the presence of aggressive optimizations is much more challenging.

In this section, we present an *Unoptimized-IR-to-Optimized-Assembly* translation validation tool, **COUNTER**[1], which can automatically compute equivalence across an unoptimized IR specification and the optimized 32-bit x86 assembly implementation of a program. COUNTER is a robust and comprehensive equivalence checking tool as it can automatically compute equivalence

[1]We use the same name for the tool as our proposed correlation algorithm

across a long and rich pipeline of transformations and optimizations at the scale of real-world programs. Further, COUNTER is a cohesive tool as it uses both an advanced (and incremental) correlation algorithm (*Counter*) to handle significant structural differences due to composition of multiple transformations and an expressive (and purely static) invariant inference algorithm (*Sifer*) to generate expressive invariants across two different syntaxes.

To our knowledge, our proposed COUNTER tool is the first demonstration of a completely automatic black-box equivalence checker that can successfully compute equivalence across the unoptimized IR and the optimized x86 assembly program. This includes the long and rich pipeline of transformations like loop unrolling, peeling, unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, etc. An online demo of the proposed equivalence checking tool COUNTER is available at [1].

We begin the discussion on COUNTER tool by discussing the implementation details in section 5.1. We present the evaluation for COUNTER tool in section 5.2 and its limitations in section 5.3. We present a detailed comparison of our tool with respect to prior equivalence checking tools in section 5.4.

5.1 Implementation Details

In this section, we present the implementation details of the tool.

5.1.1 Logical Representation

We convert the C program to a custom IR that resembles LLVM IR with the only major difference that our IR does not support LLVM's undef and poison values and instead treats all error conditions as UB. We represent the IR and x86 assembly programs as CFGs by adding a node for each PC and an edge for each possible control-flow across PCs. A CFG's machine state is a symbolic representation of the abstract machine state encoded using bitvectors to represent IR variables and x86 registers. Further, a byte-addressable array is used to represent memory.

We use our own symbolic representation which is very similar to SMT's QF_AUFBV[2] theory. The primary difference is that our custom SMT representation uses higher-level select and store operators that have an associated `sz` parameter: $\texttt{sel}_{\texttt{sz}}(\texttt{arr}, \texttt{addr})$ returns a little-endian concatenation of `sz` bytes starting at address `addr` in the array `arr`. Similarly, $\texttt{st}_{\texttt{sz}}(\texttt{arr}, \texttt{addr}, \texttt{data})$ returns a new array that has contents identical to `arr` except for the `sz` bytes starting at `addr` which have been replaced by `data` in little-endian format. These higher-level operators help us reason about equivalence of memory-regions through lightweight decision procedures that we have implemented; offloading such equivalence proof obligations to an off-the-shelf SMT solver would require reasoning about sub-arrays, which would involve universal quantifiers in our proof-obligation expressions. Instead, all our proof obligations to the SMT solver are encoded (through translation from our custom symbolic representation) in the more-tractable quantifier-free QF_AUFBV theory. Further, using our own symbolic representation allows us to codify pattern-matching simplification passes on the expressions in the proof obligations (before offloading them to the SMT solver), which helps improve performance [29]. Transfer function p_ω for edge transitions in the CFG are directly derived from the operational semantics of the programming language: most operations in the programming language translate directly to a combination of one or more operators in SMT-like algebra (e.g., + to `bvplus`, read-memory-dereference to `select`, write-memory-dereference to `store`, etc.). Similarly, edge-conditions $econd_\omega$ are derived from corresponding control-flow conditions in the original program. Edges representing procedure returns are labeled with their respective actions that encode the state-elements on which the observable event depends.

5.1.2 Deterministic CFG Construction

While the original program may be non-deterministic, the CFG needs to be deterministic (by definition). For example, the C language has several sources of non-determinism related to order of evaluation, datatype bitwidths, etc., where the compiler is free to choose one of the many possible legal behaviors. Fortunately, most such types of non-determinism in C get eliminated at the time of lowering the source language program to LLVM-like IR (which is the input to

[2]Quantifier free theory for arrays, uninterpreted functions and bitvectors.

our tool), e.g., the IR chooses a deterministic order of evaluation and a fixed data-layout for all C types. Thus, if the non-deterministic choices made by the IR are the same as the choices made by the compiler used to generate the x86 assembly, we can safely ignore these types of non-determinism.

However, non-determinism due to undefined-behavior (UB) semantics is still present at the IR level. As discussed in section 2.2.1, in order to determinize UB, we use a special `Error` node in the CFG to indicate that UB was triggered. The `Error` node has no outgoing transitions and once entered, the program stays in the `Error` node forever henceforth. To model undefined behavior, the CFG representation associates each edge with UB assumptions σ_ω, that encode the conditions that assert the absence of undefined behavior. If at the source node of edge ω, the condition σ_ω is violated, then the CFG transitions into a special `Error` node. In other words, the transition to the special `Error` node in the CFG indicates that UB was triggered. The edges that transition into the `Error` node are labeled with a special observable `Error` action. Notice that this construction always yields well-formed CFGs.

5.1.3 Observable actions

Observable actions are associated with each procedure exit edge. An observable action for an edge ω, represented using τ_ω takes as input the machine state at the start node of the edge and returns the state-elements that uniquely determine the observables. This includes the value being returned (e.g., argument to `return` instruction in (LLVM-like)IR program, `eax` for a 32-bit x86 program), and the state of the memory-regions that are live at function return (heap and memory regions for global variables).

5.1.4 Memory Model

To prove equivalence, we need to reason about relations between various state-elements of the programs which include the IR program variables, x86 assembly program registers, *memory-regions* and *stack-slots*. We explain the latter two in more detail here.

Memory-regions: The C abstract machine associates objects with memory regions; object memory can be allocated through global and local variables or through procedures that manipulate the heap (e.g., `malloc`).

In our proposed COUNTER tool, the entire program memory is modeled as a single array. But, in order to reason about compiler transformations that translate program variables to memory locations, an equivalence checking tool needs to identify and model separate memory regions for global and local variables, stack and heap. In our implementation, we model one contigous memory region for each global variable in the program. For the x86 assembly program, we identify these global variables through the symbol table in the ELF executable. The size of this memory region corresponds to the variable size and is known at program entry. We do not support Address-Taken Local variables (ATLs) and Fixed/Variable-length Local Arrays (VLAs). We use LLVM's `mem2reg` pass to promote other local variable allocations (i.e., the *LLVM alloca* instructions) to IR variables.

Further, there is no notion of local variables in x86 assembly and instead we have a continous memory region belonging to the `stack` that is used by the compiler to spill pseudo-registers. The code to manage the `stack` region is generated by the compiler. The remaining memory space, that neither belongs to one of the global variables, nor to the `stack`, is modeled as `heap`. The `heap` memory region can be potentially discontigous. A single memory region for the entire heap suffices because `malloc` and related procedures are independent of the C language. Also, although modern compilers include logic to identify if the `malloc` function refers to the standard "built-in" libc function; in our experimental setting, we disable such built-in assumptions through appropriate compiler flags to avoid having to model similar semantics in our equivalence checker and assume that the compiler does not distinguish between two different return values of the `malloc` function.

Since the program memory containing all these regions is modeled as a single array, each of these memory regions form non-overlapping sub-arrays inside this full array. Similar to prior work [15], we represent all these sub-arrays in the full array using symbolic *start* and *end* addresses. We also encode the non-overlapping constraints for these sub-arrays as described in [15].

Stack-slots: The `stack` memory region is used by the compiler to temporarily spill pseudo-registers and to pass arguments/return values for function calls. The stack memory for a 32-bit x86 assembly program can be thus split into various stack-slots, where each stack-slot represents a 8-bit, 16-bit or 32-bit value stored at a constant offset from the *input stack pointer*. The input stack pointer is the value of the *stack pointer register* at the entry to the x86 assembly program, and represents the top of the stack at function entry. To identify stack-slots, an available-expressions analysis (discussed in section 5.1.5) is performed to identify registers that hold a value that is at a constant offset from the input stack pointer. All memory accesses (i.e., select and store operations) that are at these identified constant offset addresses from the input stack pointer are then collected and are identified as the stack-slots that need to be tracked.

In this thesis, we use the term "state-elements" to collectively include the IR variables, x86 assembly registers, the memory-regions, and stack-slots.

5.1.5 Points-to and other standard data-flow analyses

Before proceeding with the proof construction, the following standard data-flow analyses are performed on the CFG representation of the specification program C and the implementation program A:

Available-expressions Analysis: It is a forward data-flow analysis that identifies the expressions that are available at a given program point or node of the CFG. The values of this data-flow analysis involve an available-expressions map from the "state-elements" to the "expressions". The transfer function of the DFA involves adding a mapping $\texttt{x} \mapsto \texttt{e}$ to the available-expressions map for a statement `x:=e` in the transfer function of an edge in the CFG. The meet operator is intersection, and the available-expressions map at program entry is initialized to an empty map.

For the implementation program A, the available-expressions analysis identifies all register values that are always at a constant offset from the stack pointer value at function entry. The identified registers are then used to identify the stack-slots in the assembly program. These stack-slots may represent the pseudo-registers that were temporarily stored (spilled) in the memory due to

a limited number of registers in the x86 target architecture. These stack-slots are considered as bitvector variables during invariant inference which enables inference of relations between the value stored at these stack locations and the corresponding variables in the IR. These stack-slots are also considered during the alias-analysis to infer precise points-to information for the values stored at these locations.

The results of the available-expressions analysis at a node are also encoded as node-invariants to supplement the invariant inference algorithm.

Points-to Analysis: It is a forward intra-procedural, flow-sensitive, field-insensitive, untyped data-flow analysis to identify the set of memory regions that a state-element *may* point to. The potential memory regions for programs without local arrays (as considered in this work) include a memory region to represent the heap, a memory region per global variable, and for the x86 program, a memory region for the stack. The values of this data-flow analysis represent the aliasing information for each state-element at each program point and involves a points-to map from a "state-element" to a "set of memory-regions" it may point to.

We use a sound and over-approximate algorithm for the points-to analysis that is similar to the algorithm described in [15, 19]. It involves identifying *C*'s *based-on* relationships (§6.7.3.1 in [34]) and tracking the flow of values. At a high-level, the transfer function involves identifying (a) if a state-element `x` is *linearly-related* to another state-element `y` (in which case the points-to set of `x` becomes the points-to set of `y`), or (b) if a state-element `x` may *depend-on* the value of another state-element `y`, in which case the points-to set of `x` is combined (set-union) with the points-to set of `y` to obtain the new points-to set of `x`. The meet operator is union. Also, it is conservatively assumed that the input arguments to a procedure can point to either the global memory regions or the heap memory region.

Since our tool does not support Address-Taken Local variables and Fixed/Variable-length Local Arrays, for the rest of the programs, our points-to analysis is able to categorize most memory accesses as either accessing only the stack region or definitely not accessing the stack region (i.e., accessing global(s) or heap). This allows us to model these two regions as separate arrays during the discharge of proof obligations, just before transmitting the proof query to the SMT solvers.

$$Inv \rightarrow \quad \{\mathbb{G}_{affine} \mid \mathbb{G}_{mem} \mid \mathbb{G}_{Cineq} \mid \mathbb{G}_{leq} \mid \mathbb{G}_{ineq}\}$$
$$\mathbb{G}_{affine} : \{\textstyle\sum c_i v_i + c_0 = 0\} \quad \mathbb{G}_{mem} : \ \{\mathtt{H_C} = \mathtt{H_A}\}$$
$$\mathbb{G}_{Cineq} : \ \{\pm v \leq 2^c\} \quad \mathbb{G}_{leq} : \ \{r_1 \leq r_2\} \quad \mathbb{G}_{ineq} : \ \{r_1 < r_2\}$$

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

Liveness Analysis: The liveness analysis is a standard backward data-flow analysis that identifies the state-elements that are live at any program point.

All three analyses discussed above are over-approximate and run in tandem within an outer fixed-point iteration loop, because it is common for one analysis to improve the results of the other, e.g., the available-expressions analysis may identify more stack slots, that may improve the points-to analysis, which may further help the available-expressions analysis to identify more stack-slots. Similarly, the points-to analysis may improve the results of the liveness analysis.

Reaching Definitions Analysis: A must-reach definitions analysis [2] is performed on C and the output of this analysis is used during invariant inference, such that only those bitvector variables are considered at a node n whose definition must reach n.

5.1.6 Invariant Inference Grammar

COUNTER tool instantiates multiple grammars as shown in fig. 5.1 for the invariant inference algorithm *Sifer*. The grammar $\mathbb{G}_{affine}$ enumerates affine invariants among the bitvector program variables represented using v_i. Here, c_i represents a bitvector constant. The bitvector program variables considered for invariant inference at a correlated PCpair (n_C, n_A) include the live registers and defined stack slots at the PC n_A for the implementation program A and all defined bitvector variables at the PC n_C for the specification program C. Considering only live variables (instead of all defined variables) for the specification program may not work for transformations involving computation re-ordering; in the presence of such transformations the live variable(s) in A may not get correlated with any live variable in C and is instead correlated with an intermediate variable(s) in C.

A similar affine invariants based grammar has been used by Churchill et al. [11] which uses this grammar in the context of Assembly to Assembly equivalence checker. The number of registers (or program variables) in an assembly program are very small as compared to the number of defined variables at a PC in a LLVM-like IR program (which is a Static Single Assignment (SSA) based representation) used as specification in our case. To achieve scalability in the presence of large number of defined SSA program variables, we use program slicing based heuristics to reduce the number of program variables to be considered for invariant inference at a product-CFG node.

The grammar $\mathbb{G}_{mem}$ represents the memory region equality predicate, where $\mathtt{H_C}$ denotes the specification program's non-temporary memory region (i.e., memory regions for heap and global variables) and $\mathtt{H_A}$ denotes the implementation program's non-temporary memory region. The grammars $\mathbb{G}_{Cineq}$, $\mathbb{G}_{leq}$ and $\mathbb{G}_{ineq}$ enumerate inequality predicates between the program variables denoted by v and between the registers in the implementation program denoted by r.

The *Sifer* algorithm involves computing the strongest invariant cover for each of these grammars. We use the first algorithm presented in section 4.2.5 to compute the strongest invariant for the affine grammar $\mathbb{G}_{affine}$. We use the second algorithm (which is a Houdini-like approach [27]) presented in section 4.2.5 to compute the strongest invariant for the rest of the grammars.

5.1.7 Discharging proof obligations

For every SMT proof obligation generated by the COUNTER tool, three off-the-shelf SMT solvers are spawned in parallel: `z3-4.8.7`, `Yices2-45e38fc`, and `cvc4-1.7`. For `unsat` results, we return as soon as the first solver finishes. For `sat` results, we opportunistically try and collect multiple counterexamples (to aid our counterexample-driven procedures): we wait for the first solver to finish; if the first solver finishes with a `sat` result in time t, then we wait till time $2 * t$ and return the counterexamples generated by all solvers that finished in time $2 * t$ (thus doubling our query times in the worst case). In addition to improving efficiency and providing more counterexamples, employing multiple SMT solvers also improves the reliability of our verifier because it allows cross-checking of the results of one SMT solver against another. In fact, we found a bug in Yices during our experiments, which was fixed immediately upon reporting [73].

To avoid SMT solver timeouts, the *simplifications* and the *query decomposition* technique detailed in [29] are used.

5.2 Evaluation

5.2.1 Experimental Setup

We evaluate **COUNTER** on a set of benchmarks involving extensive loop and vectorization optimizations, which include all the benchmarks used in the prior work on assembly-to-assembly equivalence checking [11]. In addition to the examples in figs. 2.4, 3.1 and 3.3, our evaluation involves two sets of benchmarks: the first set of benchmarks includes programs (C functions) from the TestSuite for Vectorizing Compilers (TSVC) [45], and the second set of benchmarks is a set of 27 distinct vectorizable loop patterns that we have mostly taken from the LORE repository [10]. The TSVC functions operate on global arrays of floating point values. Since the current implementation of COUNTER tool does not support floating point types, similar to prior work [11], we also systematically replace floating point types with integer types for the TSVC functions. As a result, both the functions taken from TSVC and LORE repository operate on statically-allocated fixed size global arrays of integers.

All programs are compiled using recent versions of production compilers, namely, GCC-8, Clang/LLVM-11, and ICC-18.0.3 with `-O3 -msse4.2` compiler flags to generate optimized x86 binaries. For experimental evaluation with each compiler, only those functions that are vectorized by that compiler are selected. For each of these selected compiler-function pairs, we attempt to prove equivalence across the unoptimized LLVM IR (generated by `clang -O0`) and 32-bit x86 assembly program generated by an optimizing compiler. We use a global timeout of five hours to generate the equivalence proof for the given program-pair, an SMT-solver timeout of five minutes for each SMT proof query, and a memory limit of 12GB for a single equivalence check.

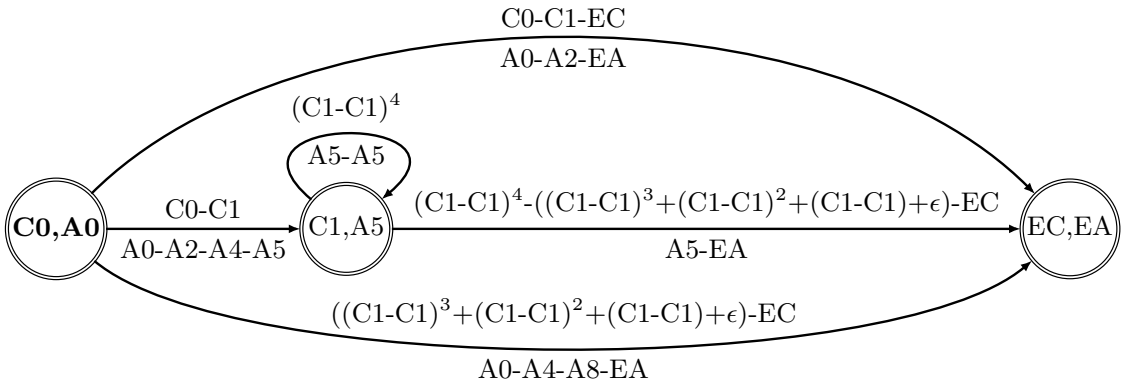
As discussed in section 3.3.1, the *Counter* algorithm takes a parameter μ_C as input to bound the length of the candidate correlations in C. In general, we find that the required value of μ_C

```
C0: void init1d(int n) {
C1:   for (int i = 0; i < n; i++)
C2:     a[i] = b[i];
EC: }
```

(a) C program

```
A0:  init1d:
A1:  r1 = 0
A2:  if (n <= 0) ret
A3:  r2 = n % 4; r3 = n - r2; r4 = 0
A4:  if (r3 == 0) goto A8
A5:    a[r4, .. r4+3] = b[r4, .. r4+3]
A6:    r4 += 4
A7:    if (r4 != r3) goto A5
A8:  if (r2 >= 1) { a[r4] = b[r4]; r4++ }
A9:  if (r2 >= 2) { a[r4] = b[r4]; r4++ }
A10: if (r2 == 3) { a[r4] = b[r4]; r4++ }
EA: ret
```

(b) (Abstracted) Assembly code



(c) Product-CFG

Figure 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.

needs to be at least twice the unroll factor used by the compiler, to be able to handle the cool-down loops in the vectorized assembly programs. This can be seen using a simple vectorization example shown in fig. 5.2. In the assembly program shown in fig. 5.2b of this example, the path from the loop head to the program exit `(A5-EA)` involves four unrolled iterations of the loop body in the C program shown in fig. 5.2a. These four unrolled iterations are followed by up to three residual iterations before reaching program exit. Thus, the loop head to the program exit `(A5-EA)` path in the assembly program will be correlated with seven iterations of the loop

in the C program (which can be captured only at $\mu_C \geq 7$) when the compiler used an unroll factor of four. Since, using a higher value of μ_C increases the total number of possible correlation candidates and hence the search space, we use $\mu_C = 2 * \mu^o$, where μ^o represents the unroll factor used by the optimizing compiler. By running our experiments with different values of $\mu_C = \{1, 2, 4, 8, 16, 32\}$, we empirically identify that both GCC and ICC perform loop unrolling with a maximum unroll factor of four, while LLVM uses an unroll factor of eight. Thus, for our evaluation results presented here, we use the value for the parameter μ_C as $\mu_C = 8$ for GCC and ICC and $\mu_C = 16$ for LLVM.

5.2.2 Results

Benchmark Categories

We divide our benchmarks into three categories:

1. 28 TSVC functions that were a part of the benchmarks evaluated by [11].
2. TSVC functions for which COUNTER is the first tool to automatically generate equivalence proofs (they were not included in [11] benchmarks).
3. Loop nest patterns taken from the LORE repository.

For TSVC benchmarks, we present results for all three compilers. For LORE loop nests, we use one representative pattern for a set of structurally-similar program/transformation pairs, irrespective of the compiler that generated it. The space of additional transformations performed in the third category of benchmarks (that are not covered by the first two categories) include loop splitting, loop fusion for bounded number of iterations, loop unswitching, and summarization of loop with small and constant bounds. The number of loops per function and maximum loop nesting depth varies between one and three for the loop patterns in this last category. Table 5.1 tabulates the results of our experiments with all these three categories of benchmarks.

Success and Failures

First set of benchmarks: For the 28 TSVC functions that were evaluated by prior work on

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

		TSVC functions demonstrated by prior work			TSVC functions not demonstrated by prior work			LORE Loop Nests			
								All loops have memory write		At least 1 loop with no memory write	
		`gcc`	`llvm`	`icc`	`gcc`	`llvm`	`icc`	$\mu^o 4$	$\mu^o 8$	$\mu^o 4$	$\mu^o 8$
	Total/Failing functions	28/1	28/0	28/3	28/7	30/4	60/22	11/0	11/0	16/0	16/0
	Avg/Max ALOC	16/44	19/51	19/40	25/64	31/72	29/95	19/28	24/53	29/48	37/100
	Avg/Max # of product-CFG nodes	3/4	3.1/5	3.2/5	3.5/5	3.4/5	3.5/6	4.4/7	4.4/7	4.8/7	5/8
	Avg/Max # of product-CFG edges	3.2/7	3.3/7	3.6/9	4/7	3.9/8	4.2/11	5.2/9	5.2/9	5.9/9	6.8/18
	Avg # of total CEs / node	17	27	16	18	28	18	16	24	20	30
	Avg # of gen. CEs / node	13	22	11	12	22	13	10	17	10	16
BFS	Avg equivalence time (seconds)	209	70	15	201	3842	110	107	2243	131	676
BFS	Avg # of paths enumerated	44	89	53	95	160	103	179	344	232	469
BFS	Avg # of paths pruned	28	45	32	49	80	54	66	98	130	251
BFS	Avg # of paths expanded	3.3	3.9	3.8	4.6	5.3	5.8	7.1	8.5	8.9	14.8
DFS	Memory/timeout reached	0	2	0	1	6	1	0	1	12	16
DFS	Avg # of paths enumerated	173	3904	315	5776	14992	2635	301	561	17518	27727
DFS	Avg # of paths expanded	35	252	52	518	913	262	111	208	4582	3781
	Avg # of paths expanded DFS/BFS	11	65	14	113	172	45	16	24	515	255

assembly-to-assembly equivalence checking [11] and across optimizations performed by the three compilers (for a total of 84 program-pairs), the COUNTER tool is able to compute equivalence proofs for all but four of these program-pairs (see row `Failing functions` of table 5.1). Three of these four failing program-pairs — `s176 compiled with GCC`, `s1112` and `s243 compiled with ICC` — involve non-bisimilar transformations, namely loop tiling and interchange, which are beyond the scope of transformations supported by the COUNTER tool. We confirm that the prior work on equivalence checking from which this benchmark is drawn [11] is also unable to compute equivalence proofs across these three program-pairs, and the reason these were reported as successful equivalence checks in their paper is because the authors used older compiler versions (which did not perform such transformations for these functions). One program-pair (`s351` when compiled with ICC) involves loop re-rolling which is out of scope for the COUNTER tool. The next row in the table 5.1 shows the average and maximum Assembly Lines of Code (ALOC) in the optimized assembly program across all these functions. It is important to note that the C source code for all these 28 functions involve only a single loop and involve no control flow within their loop bodies.

Second set of benchmarks: We next consider the remaining TSVC functions and report only those program-pairs that involve some form of vectorization in their optimized assembly code. There are 28, 30, and 60 such program-pairs (where vectorization was involved) for GCC, LLVM, and ICC compilers respectively. Of these, the COUNTER tool is able to compute equivalence for all but 7, 4, and 22 program-pairs respectively. The primary cause for equivalence failures is the presence of non-bisimilar transformations, namely loop interchange, fission, fusion, tiling, acceleration; an unbounded number of memory writes are re-ordered through such transformations and so these equivalences are difficult to establish through bisimulation relations. Only four of these failures (one each in GCC and LLVM, and two in ICC) are due to SMT solver timeouts during invariant inference. It is interesting to note that ICC is able to perform vectorization for a larger number of programs (60 vs. 30) and that most such vectorizations involve non-bisimilar transformations (20 failures).

Among the function-compiler pairs for which the COUNTER tool successfully computes equivalence for: (1) six functions contain more than one loop in function body; (2) six functions have nested loops of depth up to 2; (3) one function has both nested loops and multiple outer loops; (4) nineteen functions have control flow inside the loop body; (5) eight functions use multi-dimensional arrays potentially involving non-regular memory accesses; and (6) eleven functions have at least one loop without a memory write. Recall that correlation identification becomes easier if all the C program loops contain updates to the memory, because the pruning based on memory relations is then able to reduce the search space.

Importantly, compilations of these TSVC functions are among the most challenging program-pairs for equivalence checking because there is a large syntactic gap between the IR representation and the assembly level implementation. Further, the rich and complex pipeline of transformations from unoptimized IR to optimized x86 assembly implementation results in significant structural differences between them. The proposed tool, COUNTER, is able to automatically compute the equivalence between the IR specification and the assembly implementation for these programs because of the robustness of the *Counter* algorithm which efficiently finds the required product-CFG in an exponential search space and the incremental *Sifer* algorithm which infers expressive invariants in a scalable manner.

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

TSVC functions demonstrated by prior work								TSVC functions not demonstrated by prior work							
Name	ALOC			Name	ALOC			Name	ALOC			Name	ALOC		
	gcc	llvm	icc		gcc	llvm	icc		gcc	llvm	icc		gcc	llvm	icc
s000	11	13	15	s243	21	51	✗	s111	28	⊗	⊗	s271	⊗	43	19
s1112	12	17	✗	s251	11	15	15	s1111	20	⊗	30	s2710	⊗	⊗	44
s112	22	8	12	s3251	44	50	39	s1115	⊗	✗	28	s2711	⊗	47	21
s121	18	32	20	s351	28	17	✗	s1119	✗	14	⊗	s2712	⊗	43	19
s122	17	17	24	s452	14	19	18	s113	20	⊗	23	s272	⊗	⊗	26
s1221	9	8	13	s453	11	13	15	s114	⊗	⊗	50	s273	⊗	53	25
s1251	13	12	17	sum1d	15	16	18	s116	⊗	17	⊗	s274	⊗	⊗	23
s127	17	18	23	vdotr	17	19	21	s1161	⊗	⊗	46	s276	⊗	⊗	29
s1281	17	15	21	vpv	9	11	13	s119	27	31	28	s293	⊗	⊗	13
s1351	9	11	13	vpvpv	10	13	14	s1213	⊗	⊗	37	s311	15	15	19
s162	43	37	40	vpvts	12	15	16	s124	18	24	20	s3111	19	20	24
s173	9	8	15	vpvtv	10	13	14	s125	24	20	25	s319	22	30	27
s176	✗	21	22	vtv	9	11	13	s1279	⊗	47	22	s352	⊗	22	⊗
s2244	24	47	28	vtvtv	10	13	14	s128	20	⊗	23	s4115	⊗	⊗	32
								s131	15	29	20	s421	25	48	29
								s132	28	43	26	s423	33	46	29
								s1421	24	25	40	s441	28	⊗	34
								s171	⊗	29	✗	s442	⊗	⊗	49
								s174	64	35	52	s443	17	⊗	25
								s2233	39	✗	✗	s471	28	26	✗
								s252	⊗	18	⊗	va	8	9	12
								s253	⊗	⊗	24	vbor	✗	✗	95
								s254	⊗	8	12	vif	⊗	72	17

Table 5.2 lists out the TSVC functions along with their assembly lines of code (ALOC) belonging to both the categories discussed above for which equivalence could be established by the COUNTER tool.

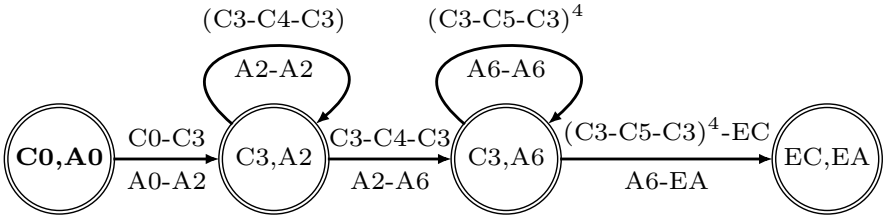
Third set of benchmarks: This set of benchmarks includes 16 different loop nest patterns. For each of these 16 patterns, we test two variations: one where the loop bodies involve a memory write, and another where at least one of the loop bodies does not involve a memory write. Among the 16 variations that involve a memory write in the loop bodies, the compilers produce non-

```
C0: int loopSplitting() {
C1:   int sum = 0;
C2:   int mid = LEN/2;
C3:   for (int i=0; i<LEN; i++) {
C4:     if (i<mid) sum += c[a[i]];
C5:     if (i>=mid) sum += b[i];
c6:   }
C7:   return sum;
EC: }
```

(a) C program.
LEN is a positive multiple of 4.

```
A0: loopSplitting:
A1:  r1 = 0; r2 = 0;
A2:    r2 += c[a[r1]]
A3:    r1++
A4:    if (r1 != mid) goto A2
A5:  r1 = &b[mid]; r3 = &b[LEN]; xmm0 = 0
A6:    xmm0 += *r1, ..,  *(r1+12)
A7:    r1 += 16
A8:    if (r1 != r3) goto A6
A9:  xmm0 += (xmm0>>8) // shift right by 8 bytes
A10: xmm0 += (xmm0>>4) // shift right by 4 bytes
A11: r2 += xmm0[31:0]
EA: ret r2
```

(b) (Abstracted) Assembly code



(c) Product-CFG

Figure 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

bisimilar transformations for five of them. Thus we show results for 11 loop nest patterns where loop bodies have memory writes, and 16 loop nest patterns where the loop bodies don't have memory writes. Further, for each loop nest variation, we test across two different unroll factors ($\mu^o = 4$ and $\mu^o = 8$). The patterns with unroll factor 8 are due to compilations generated by LLVM or by GCC with the appropriate pragma switch. The COUNTER tool is able to compute the required equivalence proof for all these (16+11)*2=54 program-pairs.

Most of these source programs (16 out of 27 total) have multiple loops with potential nesting (and different variables in each loop). We find that the data-driven SPA algorithm [11], which is the closest competing algorithm to our tool in terms of capability would not be able to guess

the required alignment predicate in these cases using the restricted grammar proposed in that work. Moreover, 17 benchmark programs use multi-dimensional arrays which are out of scope for most of the prior work on equivalence checking. Six benchmark programs have control flow inside the loop body for both source and generated assembly program, and it may be difficult and expensive to identify execution traces with adequate code coverage in such programs (as required by most of the prior work on equivalence checking). As an example of the complexity of transformations involved in this set of benchmarks, fig. 5.3 shows a program-pair involving multiple transformations including loop splitting, loop unswitching, unrolling, and vectorization. The product-CFG generated by the COUNTER tool for this program-pair is shown in fig. 5.3c.

Equivalence Checking Statistics

Table 5.1 also shows the average and maximum number of nodes and edges in the product-CFG generated by the COUNTER tool for each benchmark category. Further, it shows the average number of counterexamples per final product-CFG node (`Avg # of total CEs/node`) and the average number of counterexamples that were generated (not propagated) per node through SMT queries (`Avg # of gen. CEs/node`). These counterexamples are used by both the correlation algorithm *Counter* and the invariant inference algorithm *Sifer*. Note that the number of paths in a pathset correlated in the final product-CFG can be exponential in the unroll factor (i.e., 8 and 16 for GCC/ICC and CLANG respectively), but the number of counterexamples required (ranging from 16 to 30) are very few as compared to the number of paths. In contrast, the data-driven techniques would require exponential number of traces to generate the required product-CFG for such programs.

Similarly, the number of candidate invariants that can be enumerated through the grammars (shown in section 5.1.6) that are used to enumerate the node-invariants in the COUNTER tool are huge (potentially exponential) but the total counterexamples required to infer the strongest invariant cover for such grammars are small in general (ranging from 16-30 for these benchmarks) and bounded in worst-case. *Sifer* algorithm involves generating these counterexamples on-demand as the satisfying model generated by the SMT solvers for intermediate not-provable queries and contain specific concrete mappings for the state-elements which are hard to construct using random input generation algorithms or model-checking techniques.

In table 5.1, the first row in category `BFS` (which stands for best-first search) lists the average time taken to generate an equivalence proof in each benchmark category (`Avg equivalence time`). This average equivalence checking time is mostly dominated by the invariant inference algorithm's time, which is called at each incremental step (ranging from on average 3 to 15 steps per program in a benchmark). Due to the incremental characteristic of the *Sifer* algorithm, the average equivalence checking time for a function, even after calling the invariant inference algorithm multiple times is small (as small as 15 seconds for a program-pair in the first category of benchmarks for ICC compiler).

The next three rows demonstrate statistics for the best-first search (BFS) algorithm: we list the number of correlation possibilities that were created (`paths enumerated`) before the complete product-CFG was found, the number of correlation possibilities that were remaining after pruning (`paths pruned`) and the number of correlation possibilities which were actually expanded further (`paths expanded`). This last metric is a measure of the effectiveness of *Counter* algorithm's ranking strategy : the table shows that the average number of paths expanded is small, and usually close to the average number of total product-CFG edges. Because each time a product-CFG correlation is expanded, we add an edge to the product-CFG: this confirms that in most cases, the correct correlation is ranked and picked first at each step of the backtracking search. In other words, the ranking strategy ensures that there is minimal backtracking, if any.

Comparison with a Static Strategy: The three rows labeled `DFS` (for depth-first search) in table 5.1 demonstrate the results of equivalence checking using a backtracking-based strategy relying on the static heuristic, where counterexample-guided pruning and ranking is omitted. This static backtracking strategy resembles the depth-first search approach used in [14]. In this search one part of the search tree is exhausted (depth-first) before another part of the subtree is attempted. We find that the average number of paths expanded in DFS is up to 515x more than the average number of paths expanded in BFS (last row in table 5.1); this is evidently due to the extra backtracking that occurs in the DFS strategy. In fact, the DFS strategy runs out of either time or memory resources for 39 of the 219 program-pairs for which BFS is able to successfully establish equivalence (`Memory/timeout reached`). It is worth noting that these improvements produced by the *Counter* algorithm's pruning and ranking strategies are more pronounced in programs involving loops which do not update memory in their loop bodies. For loops that update

```
C1: void *memccpy(void *dst, const void *src,
                  int c, size_t count) {
C2:   char *a = dst;
C3:   const char *b = src;
C4:   while (count--) {
C5:     *a++ = *b;
C6:     if (*b == c)
   // missing (unsigned char) type casts;
   // (*b) will get sign-extended before
   // comparison and thus may not be
   // equal to c when sign bit is set
C7:         return (void *)a;
C8:     b++;
C9:   }
C10:   return 0;
C11: }
```

(a) `memccpy` function of diet libc

```
const char src[] = { 255, 128 };
char dst[2] = { 'A', 'B' };
memccpy(dst, src, 255, 2);
if (dst[1] != 'B')
  printf("BUG!")
```

(b) Sample input for triggering the bug

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

memory in their bodies, the `InvRelatesMemAtEachNodes()` check (in section 3.3.1) allows early backtracking in situations where an incorrect correlation is chosen.

Other Explorations

In addition to these benchmarks, we have applied the COUNTER tool for verifying equivalence of several benchmarks including all the examples used in previous papers on equivalence checking [11, 14, 36]. We have also applied the COUNTER tool to verify *libc* string functions implementations, and in one such experiment, we compared the OpenBSD [54] libc implementation against diet libc [20]. Through this exercise, we uncovered three subtle and serious bugs in diet libc implementation, one of them shown in fig. 5.4; these bugs were acknowledged and fixed by diet libc developers immediately upon reporting. All of the three bugs were related to missing type casts in the C code. Surprisingly, these bugs had escaped years of testing and deployment.

Comparison of SMT Solvers for Counterexample Generation for Invariant Inference: Recall that the COUNTER tool uses three off-the-shelf SMT solvers — Z3, Yices2, and CVC4 — that execute

in parallel to discharge each SMT proof obligation. It is interesting to note that different SMT solvers exhibit significantly different behavior in our experiments: while Yices2 is usually much faster at discharging SMT queries (around 98% of queries are first answered by Yices2), the other two solvers actually produce "better" counterexamples (satisfying assignments) for our equivalence procedure.

To see this with an example: let's say at some node n in the partial product-CFG, there exist two *unconstrained* and *independent* 32-bit variables `x` and `y`. Also assume that we obtain $(0, 0)$ (short for $\{\texttt{x} \mapsto 0, \texttt{y} \mapsto 0\}$) as the first satisfying assignment (counterexample) through an incoming edge at n. The strongest affine-invariant cover inferred by the *Sifer* algorithm for this counterexample will be $(\texttt{x} = 0) \wedge (\texttt{y} = 0)$. Now assume that the next query to the SMT solver generates an assignment that does not satisfy this inferred invariant, and let's say we get another counterexample $(0, 2^{31})$. With these two counterexamples obtained so far, the strongest affine-invariant cover will now be weakened to $(\texttt{x} = 0) \wedge (\texttt{2y} = 0)$. Repeating this, let's assume that the third counterexample that we obtain through the SMT solver query is $(0, 2^{30})$; now, the new affine-invariant cover would become $(\texttt{x} = 0) \wedge (\texttt{4y} = 0)$. This pattern of counterexamples where the satisfying assignments are successively decreasing powers of 2 can potentially go on — notice that for this pattern of counterexamples returned by the SMT solver, we would require 64 SMT queries (32 queries for different values of variable `x` and 32 queries for variable `y`) before reaching the desired invariant, i.e., `True` (recall that `x` and `y` are unconstrained and independent). On the other hand, if the SMT solver had returned counterexamples $(0, 0)$, $(3, 5)$, and $(5, 7)$ in the first three SMT queries, we would have inferred the required invariant `True` within just three queries. Thus, the speed (or the number of steps in the fixed-point transfer function f_ω) of the invariant inference algorithm, *Sifer*, also depends on the "quality" of counterexamples returned by the SMT solver.

It turns out that Yices2 is more prone to the former behavior (returning counterexamples that involve decreasing powers of 2), even though it is faster in discharging the individual queries than Z3 and CVC4. This observation motivated our opportunistic counterexample collection scheme described in section 5.1.7, wherein we opportunistically try to collect counterexamples from multiple solvers for `sat` results.

```
C1: if (a)                          A1:  mn = a ? m : n;
C2:   while (i < m) S;              A2:  while (i < mn) S;
C3: else
C4:   while (i < n) S;
```

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

5.3 Limitations

The proposed Unoptimized-IR-to-Optimized-Assembly Translation Validator, COUNTER, is based on the *Counter* algorithm for identifying the required correlation and the *Sifer* algorithm for inferring expressive invariants. We discuss the limitations of both these algorithms here.

5.3.1 *Counter* Algorithm

The proposed *Counter* algorithm is not without limitations. We list the major limitation below:

1. For the example shown in fig. 5.5 the two near-identical loops in C are transformed into a single loop in A. Here, the path `(A1-A2)` in A needs to be correlated with two distinct pathsets in C: `(C1-C2)` and `(C1-C4)`. Notice that this violates the Observation-C (section 3.1) because the required pathsets in C have different endpoints, `C2` and `C4`. Because *Counter* algorithm only correlates a pathset in A with a single pathset in C (section 3.3.5), it will be unable to identify the required product-CFG in this case. In other words, while *Counter* supports path specialization transformations, in its current form, it cannot tackle path merging transformations. It is possible to relax the correlation condition used by the *Counter* algorithm and allow a pathset in A to correlate with multiple pathsets in C. Such choices should carefully balance the algorithm's common-case running time against its ability to handle these corner cases.

2. In its current form, the *Counter* algorithm is only interested in identifying bisimulation relations and a whole class of transformations that do not preserve program structure (*aka*

non-bisimilar transformations) are out of scope for the algorithm. Some examples of non-bisimilar transformations are loop fusion, loop fission, loop interchange, and loop tiling.

Prior work by Verdoolaege et. al. [71] can handle such transformations and is based on summarizing the loops with a polyhedral model and comparing the polyhedral models for the given unoptimized and optimized program-pair. However, it only supports affine programs and the optimized code generated through compilers would usually contain several logical, shift, branching, bit-manipulation, load-store, function-call, etc. opcodes, none of which would fit in their framework of affine programs, and they would usually report equivalence failures for such programs. Our work on the other hand is limited to bisimilar transformations but supports a general class of program-pairs.

3. Finally, *Counter* is unable to compute equivalence if the unrolling performed in the compiler transformation is out of range of the μ_C value considered in the algorithm. Larger μ_C values result in larger sets of possible correlations at each step, and thus potentially make the equivalence checking algorithm slower. As we discuss in our experiments, we find that $\mu_C = 16$ suffices for the transformations produced by the recent versions of GCC, LLVM, and ICC compilers (specific versions are mentioned in section 5.2), but may not suffice for the unrolling performed by the future versions of the compilers. We must point out that *Counter* only unrolls paths in C through μ_C, and does not unroll paths in A — this may be inadequate for computing equivalence across certain types of loop re-rolling transformations.

5.3.2 *Sifer* Algorithm

The major limitations of the proposed invariant inference technique are:

1. The proposed *Sifer* algorithm, as shown in grammar in fig. 5.1, computes only equality invariants between the memory regions. This cannot handle transformations like store sinking, which result in unequal memory states at the product-CFG nodes. An example program-pair involving the store sinking optimization is shown in fig. 5.6. Here, the C program shown at the left writes the updated value to memory region corresponding to the global variable `sum` in each loop iteration, whereas the assembly program shown at the

```
int sum, a[LEN];
C0: void storeSinking() {
C1:   sum = 0;
C2:   for (int i=0; i<LEN; i++) {
C3:     sum += a[i]
c4:   }
C5: }
```

(a) C program.The variable `sum` and array `a` are global variables.

```
A0: storeSinking:
A1:  r1 = 0; r2 = 0;
A2:    r2 += a[r1]
A3:    r1++
A4:    if (r1 != LEN) goto A2
A5:  mem[&sum] = r2
A6:  ret r2
```

(b) (Abstracted) Assembly code after store sinking optimization

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

right updates the value in the register `r2` in the loop and updates the memory region corresponding to the variable `sum` (denoted by `mem`[&`sum`]) at the end of the loop. Thus, due to store sinking optimization the equality relation between the memory regions corresponding to the global variable `sum` would not hold at the loop head or at any other PC in the loop.

A more expressive grammar and a more sophisticated but efficient memory invariant inference algorithm would be required in this case to handle such transformations which result in unequal memory states at the product-CFG nodes. Also, similar techniques are required to handle transformations involving non-linear (including polynomial) invariants.

2. Transformations like code hoisting, software pipelining and loop invariant code motion involve moving computations, that happen later in the specification program, ahead in the implementation program. We currently use ad-hoc techniques involving use of lookahead expressions which work for limited program structures and transformations only.

.

5.4 Comparison with Prior Translation Validation Tools

The notion of translation validation as a technique for verifying the correctness of the compilations produced by the optimizing compilers was introduced by Pnueli et al. [56]. Rather than verifying in advance that the compiler always produces a semantic-preserving executable code (vis-a-vis input specification source code), translation validation involves verifying the equivalence between the input specification source code and the output executable code for every compilation.

In the context of compilation of imperative programming languages, the input specification represents the language-level semantics of the program, e.g., the C language semantics of a C program, and the executable implementation models the semantics of the low-level assembly opcodes that run directly on the hardware, e.g., the semantics of the x86 assembly instructions. Thus, equivalence in this setting needs to be computed between the C program specification and the x86 assembly program implementation. Most prior work on translation validation has usually simplified the problem by computing equivalence only across a cross-section of the intermediate stages of the translation pipeline and can be broadly divided into following categories:

IR-IR Equivalence Checkers: The prior work on equivalence checking in this category [52, 69, 25, 75, 44] verifies an IR implementation (treated as the program specification) against another IR implementation, which is at a higher abstraction level than the actual implementation (e.g., x86 assembly).

One of the earliest IR-IR equivalence checking effort is by Necula et al. [52]. They presented a translation validation infrastructure (TVI) to verify the equivalence across a few selected IR transformation passes (like branch optimization, common subexpression elimination, register allocation and code scheduling) in GCC. TVI constructs a simulation relation to establish equivalence across the input and output IR for each of these selected transformation passes. It uses branch heuristics for correlating the program fragments and cannot handle several structure altering transformations like replacement of branches with conditional moves or other structure changing loop transformations. Further, the weakest precondition based invariant inference used in TVI for computing invariants for this pass-by-pass equivalence checking cannot be used for program-pairs that are generated by a composition of multiple transformations.

A pass-by-pass approach for verifying IR-level transformations has also been proposed by Tristan et al. [69]. They propose semantic-preserving rewrite rules (similar to the transformation rules used by an optimizing compiler) to transform the *value graph* of input and output IR programs. If the resulting normalized value graph for the input and output IR programs (after applying the rewrite rules) match, then the equivalence is proven. They demonstrate this approach to prove equivalence across LLVM compiler transformation passes like advanced dead code elimination, global value numbering, sparse-condition constant propagation, loop invariant code motion, etc. A similar technique, based on applying equality preserving transformations on Program Expression Graphs (PEGs) was proposed by prior work on *equality saturation* [67, 64]. They demonstrated their technique using a tool called Peggy, which as compared to prior work [69], additionally supports a few more optimization passes like constant propagation, partial-redundancy elimination, and basic block placement. However, these techniques are limited by the rewrite rules available to the translation validator. Further, they do not support many loop transformations like loop unrolling or loop inversion because formulating rewrite rules for such transformations is non-trivial.

Zhao et al. [75] proposed a proof technique to verify the correctness of IR-level compiler transformations that rely on the SSA (i.e., Static Single Assignment) property of the IR. An example of such transformation is the `mem2reg` pass that is responsible for promoting local variables stored in memory to registers. A variant of this `mem2reg` pass in LLVM (i.e., `vmem2reg`) was proposed and verified by Zhao et al. The verification involves proving the correctness of each step (or "micro" transformation) involved in the proposed `vmem2reg` pass using COQ proof assistant and composing the proofs to establish the equivalence of the complete transformation pass. This is analogous to the prior work on verified compilers [42] which involve verifying the correctness of each individual pass manually using a proof assistant.

Another important line of work involving IR-IR equivalence checking has been proposed to verify the peephole optimizations in a compiler. Peephole optimizations perform algebraic simplifications for a sequence of IR-instructions to enable further optimizations and improve efficiency. It is very hard to manually reason about the interactions of these algebraic simplifications with the undefined behaviors present in the IR (for example poison and undef values in LLVM IR) and hence potentially result in compiler bugs. Alive [44] is an automatic formal verification tool

(along with a specification language) to verify the peephole optimizations in LLVM in the presence of undefined behavior. It involves specifying the peephole optimization to be verified in its proposed specification language and then using a SMT solver to prove its correctness. Along with verifying the specified peephole optimization, Alive can also generate the high-level (C++) code for the optimization, if found correct. Similarly, Alive-FP [47] was proposed to automatically verify the peephole optimizations involving bit-precise floating point operations in LLVM. This involves adding floating point support to the specification language proposed by Alive [44] and SMT modeling for the bit-precise floating point operations. Both Alive and Alive-FP have identified several miscompilation bugs in LLVM's *InstCombine* and *InstSimplify* passes and are used by the compiler developers to verify new peephole optimizations. Alive2 [43] was proposed later to verify other common intra-procedural IR-level transformations in LLVM. It uses bounded translation validation to handle programs with loops and has uncovered several refinement bugs across complex transformation passes including vectorization.

In comparison to these tools, COUNTER does not model the non-determinism (i.e poison and undef values) in LLVM and only models the UB assumptions for the C-language. Further, the target application for the COUNTER tool is to verify the (almost) end-to-end compilation from unoptimized IR to optimized assembly across a rich and complex pipeline of transformations and can be used for equivalence checking applications like program synthesis or superoptimization. This is different from the target application of these IR-IR equivalence checkers [44, 47, 43] which attempt to identify bugs in each individual IR-level transformation (these bugs may or may not result in an end-to-end miscompilation). Unlike the COUNTER tool, the proposed Alive2 cannot be used for program synthesis or superoptimization because Alive2 is unsound on two counts: 1) it uses an imprecise modeling for non-determinism (as explained in section 3.2 in [43]); 2) it only verifies the equivalence for bounded number of iterations for loops.

Assembly-Assembly Equivalence Checkers: Prior work in this line of research has addressed the computation of equivalence between two different assembly implementations (typically unoptimized vs. optimized) of the same program [63, 14, 60, 11].

Data-driven techniques [63, 60, 11] in this category execute both the unoptimized and the optimized assembly programs on real inputs and use the execution traces for both correlating the pro-

gram transitions and for invariant inference. These data-driven techniques require high-coverage execution traces (which can potentially be exponential in the size of the programs) to complete the equivalence proof. In comparison to this, COUNTER, is a static equivalence checking tool and it generates the required concrete models on-demand using the SMT solver queries. Also, our tool computes equivalence across two different syntaxes (IR vs Assembly) – generating the execution traces required by data-driven techniques for programs at different levels of abstractions is riddled with engineering subtleties and has not been demonstrated yet in any prior work.

A static assembly-assembly equivalence checker that does not rely on any execution data, was proposed by Dahiya et al. [14]. It uses an incremental algorithm for simulation relation construction and relies on the equality of path conditions to restrict the search space of possible correlations in the absence of execution traces. The correlation criterion proposed in the *Counter* algorithm, which is used by the COUNTER tool for correlating program transitions, is much more general as compared to the path condition equality based correlation criterion used in this prior work. As a result, the COUNTER tool can handle a larger set of transformations (like like loop splitting or all transformations that involve code specialization like loop unswitching) that result in structural differences between the input and output programs. Further, Dahiya et al. used an enumeration based invariant inference algorithm which cannot be used in a scalable manner to infer the affine invariants which are usually required to relate the state-elements of the high-level IR representation and the low-level assembly implementation. In contrast, the *Sifer* algorithm used in the COUNTER tool can infer affine-invariants in an efficient manner.

IR-Assembly Equivalence Checkers: The biggest shortcoming of an IR-IR equivalence checker is that only a subset of the transformations is verified in this approach, e.g., some of the most involved steps in code generation, such as assembly-level instruction selection are completely missed in these checkers. These ignored low-level code-generation steps are especially complex for CISC architectures that contain a large number of opcodes with rich semantics (e.g.,vector-instruction in x86). Also, these low-level transformations often contain more bugs and are thus critical for verification [72].

The assembly-assembly equivalence checkers intend to address the limitations of IR-IR equivalence checkers, and cover a large part of the compiler transformation pipeline including the

code-generation and optimization pipelines for complex x86-like CISC architectures. But, the disadvantage of these equivalence checkers is that the input program specification to these tools is in a "lowered" assembly form. A higher level specification (like IR) contains more information to detect the undefined-behavior (UB) and the lowered assembly representations lose this crucial information. To see this with an example, consider a simple C language for-loop:

$$for\ (int\ i = 0;\ i < n + 1;\ i++)\ \{...\}$$

It is a common optimization to convert such a program pattern to:

$$for\ (int\ i = 0;\ i \leq n;\ i++)\ \{...\}$$

However, this transformation is not legal unless we know that the variable n has a signed-integer type, and hence arithmetic-overflow on variable n is an undefined behavior (UB) as per C-language semantics. Through empirical evaluation, previous work on equivalence checking [44, 15] has shown that compilers heavily employ UB assumptions while making code transformations, and an equivalence checker using the assembly program as specification would result in false equivalence failures in these scenarios.

In comparison to prior IR-IR equivalence checkers and assembly-assembly equivalence checkers, our proposed tool COUNTER is a more comprehensive (almost end-to-end) and robust equivalence checker. It computes equivalence between a high-level program representation (IR) that preserves type information and a low-level optimized assembly implementation potentially involving complex instruction opcodes. This includes the long and rich pipeline of transformations like loop unrolling, peeling, unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, etc.

To our knowledge, the only other IR-Assembly equivalence checker has been demonstrated by Kasampalis et al. [35]. They propose an equivalence checker, KEQ, that can compute equivalence across the *Instruction Selection* compiler pass which takes an LLVM IR program as input and generates an assembly program as output. The key idea used by KEQ involves dividing the translation validation system into a transformation-independent module that is parameterized to

the formal semantics of the input and output language and a transformation-dependent module that is responsible for generating the proof obligations for establishing the equivalence. They also formalized a weak bisimulation notion called *cut-bisimulation* which involves matching the state-elements of the input-output programs only at selected synchronization (or cut) points. The authors claim the proposed cut-bisimulation to be more general as compared to other simulation relation variants like stuttering simulation used by prior work on witness generation [51]. The notion of simulation relation used by the COUNTER tool involves identifying correlated pathsets (which represents a set of program paths). This is analogous to the proposed cut-bisimulation, as both involve matching states only at selected PCpairs (or synchronization points). Further, we demonstrate the pathset correlation based simulation relation used by the COUNTER tool to verify transformations like loop unrolling which can not be handled by the proposed cut-bisimulation relation used by KEQ in its current form.

Cohesiveness of the Validator: The most common approach to equivalence checking involves constructing a simulation relation across the given program-pair and involves two main sub-procedures – 1) A correlation procedure that constructs the product-CFG by correlating program transitions across the specification and the implementation programs in lock-step; 2) An invariant inference procedure to identify inductively-provable invariants at each node of the product-CFG. Prior work on equivalence checking [26, 25] has shown improvements in only one of these sub-procedures in isolation without providing insights on the cohesion of these improvements with the other sub-procedure. They have proposed automated tools based on constraint-solving to prove equivalence across two structurally similar programs. As noted in section 5.1 in the paper on regression verification [26], they manually modify the programs to make them structurally similar before applying their tool. We find that using these constraint-solving based techniques along with the automatic correlation algorithms (like *Counter* or [11]) is non-trivial and has not been demonstrated yet. In contrast, the proposed tool COUNTER uses both a robust correlation algorithm (*Counter*) that works for structurally significantly different program-pairs and an invariant inference algorithm (*Sifer*) that is both efficient to be used with robust correlation algorithms and powerful enough to infer expressive invariants across complex compiler transformations.

Black-Box Equivalence Checking: A contrasting technique to the black-box equivalence checking (which makes minimal assumptions on the exact nature of transformations performed)

involves modifying the compiler source code to either produce a witnesses (or proof) during compilation itself [51] or to generate hints that can be used by a translation validator to construct the required proof [35]. Access to the compiler source code or compiler generated hints simplifies the proof construction because the exact semantics of the transformation being performed are known to the validation tool and can be used to trivially correlate the program state-elements and transitions. The biggest shortcoming of this technique is its limited applicability to closed source compilers like Intel C Compiler (ICC). In contrast, COUNTER is a black-box equivalence checking tool and does not have any such limitation. The evaluation results of the proposed tool, COUNTER, for benchmarks compiled using ICC are presented in this thesis. Further, this approach for verification which requires access to the information about the transformation being performed can not be used for constructing an equivalence checker for the application of program synthesis/superoptimization; such an equivalence checker should be agnostic to the transformations performed while proposing the (randomly chosen) optimized program. Also, note that verification using the compiler generated witness or hints has been demonstrated for only selected transformation passes for chosen compilers and involves repetitive and potentially non-trivial manual effort to implement it for other passes and compilers. On the other hand, a black-box equivalence checker like COUNTER is capable of deciding equivalence across a larger and general space of transformations. Further, it also has a significant scope of automation and can be reused to verify common off-the-shelf (COTS) compilers.

Chapter 6

Conclusion

The problem of equivalence checking, i.e. formally verifying the functional equivalence between a program specification and its implementation has several important applications. A primary application appears in the context of translation validation, which attempts to automatically generate a proof of equivalence across the transformations (translations) performed by an optimizing compiler. This involves constructing an equivalence checker which after every run of the compiler verifies the functional equivalence between the input source code and the generated executable code.

The general equivalence checking problem is undecidable and is very challenging in the translation validation context due to large syntactic gap between the source and assembly representation and the large 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 thesis proposes algorithms that help make significant progress in the space of automatic translation validation.*

Chapter 3 presents a robust correlation algorithm, *Counter*, to identify correlated program transitions (represented as product-CFG) between program-pairs that may have significant structural difference across them. *Counter* uses an incremental approach for constructing the required product-CFG, where an edge is added at each step to the partial product-CFG constructed so far.

It is based on a *best-first search* procedure that uses counterexample-guided pruning to reduce the search space of candidate product-CFGs and counterexample-guided ranking to prioritize remaining correlation candidates.

Chapter 4 presents a scalable yet static invariant inference algorithm, *Sifer*, to infer precise and expressive invariants between programs that have large syntactic gap across them. *Sifer* does not depend on any execution data, but generates concrete assignments (aka counterexamples) on-demand using SMT solver queries. It is implemented as a data-flow analysis and thus is incremental in itself. It is parameterized using an input grammar $\mathbb{G}$ such that an invariant is formed by conjuncting atomic predicates drawn from $\mathbb{G}$. An important property of *Sifer* algorithm is that, for grammars $\mathbb{G}$ with finite height, it infers the strongest inductive invariant cover in bounded number of steps. Both the incremental property and the bounded runtime, make the *Sifer* algorithm efficient enough to be used with advanced and incremental correlation algorithms (like [14] or the proposed *Counter* algorithm). Unlike single-step correlation algorithms which call the invariant inference procedure once after completing the correlation for whole program, an incremental correlation algorithm invokes the invariant inference procedure multiple times at each incremental step.

Using these algorithms, an *unoptimized-IR-to-optimized-Assembly* equivalence checking tool, COUNTER is presented in chapter 5. To our knowledge, COUNTER is the first black-box equivalence checker which can automatically compute equivalence across the long (almost full) and rich pipeline of transformations from the unoptimized IR to the optimized x86 assembly program generated using aggressive compiler flags with general purpose compilers like GCC, CLANG and ICC. This includes optimizations like loop unrolling, peeling, unswitching, versioning, loop inversion, vectorization, register allocation, code hoisting, strength reduction, dead code elimination, etc. We evaluate the COUNTER tool on vectorized benchmarks taken from TSVC suite and LORE repository.

A competing approach to translation validation for verifying the compiler correctness is certified compilation (exemplified by the CompCert compiler [42]), which involves manually developing the compiler from scratch along with the correctness guarantees using a proof assistant. As compared to translation validation which involves verifying the input-output behavior for every compila-

tion, certified compilation involves proving in advance that the compiler will always produce a semantic preserving executable code, which is a huge manual effort and is non-trivial for many loop optimizations. In comparison to this, a robust and comprehensive translation validation tool (like COUNTER) has significant scope for automation and can be reused to verify common off-the-shelf (COTS) compilers.

The proposed tool is categorized as *black-box equivalence checker* because it makes minimal assumptions on the exact nature of transformations performed. Such an equivalence checker is capable of deciding equivalence across a larger and general space of transformations. Unlike witness-generation based techniques that modify the compiler source code and puts undue burden of verification on compiler developers, the black-box equivalence checker can be built independently by formal verification experts. A robust black-box equivalence checker has high applicability in other applications and significant scope for automation. For instance, a black-box equivalence checker can be used for program synthesis, which requires the equivalence checker to consider the transformations performed while proposing the optimized program as black-box. Further, other than verification based applications like relational verification, program synthesis etc., it can be used for other applications that involve analysis of binary executable code. For instance, our proposed unoptimized-IR-to-optimized-Assembly equivalence checking tool, COUNTER, has been used by recent work on improving the debug information using the proof generated by the equivalence checker [39].

`Future scope of work:` Both the proposed correlation algorithm and the proposed invariant inference algorithm heavily use the counterexamples generated through SMT solver queries and this counterexample processing time can become a scalability bottleneck. Further, during an equivalence check, most of the time is spent in discharging SMT proof obligations. Program slicing is, although effective in reducing the number of program variables for invariant inference, but high arity intermediate predicates can still appear and may lead to SMT solver timeouts as well. We believe that future work towards scalable counterexample processing and towards making the proof effort more efficient would enable the proposed equivalence checker, COUNTER, to scale to larger programs and across more complex transformations.

Further, in its current form, the COUNTER tool does not support the non-bisimilar transforma-

tions and a tool which can automatically verify these transformations has applications in auto-parallelization and optimization. Also, automatically verifying the floating point optimizations has applications in verifying the scientific computation programs which rely on the `fastmath` floating point support and forms an important line of future work for equivalence checkers.

Another interesting line of future work is to generalize the proposed equivalence checking tool COUNTER to handle LLVM UB (similar to [44, 47, 43]). The resulting equivalence checker can then be used to verify equivalence across IR-level transformations for programs with loops in a more precise manner as compared to the bounded translation validation used by prior work.

Bibliography

[1] *[ONLINE-DEMO] Online demo of the equivalence checker*. https://compiler.ai/demo/. 2020.

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[3] Sorav Bansal and Alex Aiken. "Automatic Generation of Peephole Superoptimizers". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 394–403. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168906. URL: http://doi.acm.org/10.1145/1168857.1168906.

[4] Sorav Bansal and Alex Aiken. "Binary Translation Using Peephole Superoptimizers". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 177–192. URL: http://dl.acm.org/citation.cfm?id=1855741.1855754.

[5] Gilles Barthe, Juan Manuel Crespo, and César Kunz. "Relational Verification Using Product Programs". In: *Proceedings of the 17th International Conference on Formal Methods*. FM'11. Limerick, Ireland: Springer-Verlag, 2011, pp. 200–214. ISBN: 9783642214363.

[6] Nick Benton. "Simple Relational Correctness Proofs for Static Analyses and Program Transformations". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. Venice, Italy: Association for Computing Machin-

ery, 2004, pp. 14–25. ISBN: 158113729X. DOI: 10.1145/964001.964003. URL: https://doi.org/10.1145/964001.964003.

[7] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 184–190. ISBN: 978-3-642-22110-1.

[8] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'11. Austin, TX, USA: Springer-Verlag, 2011, pp. 70–87. ISBN: 9783642182747.

[9] Jia Chen et al. "Relational Verification Using Reinforcement Learning". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360567. URL: https://doi.org/10.1145/3360567.

[10] Z. Chen et al. "LORE: A loop repository for the evaluation of compilers". In: *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 2017, pp. 219–228.

[11] Berkeley Churchill et al. "Semantic Program Alignment for Equivalence Checking". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: ACM, 2019, pp. 1027–1040. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314596. URL: http://doi.acm.org/10.1145/3314221.3314596.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-24730-2.

[13] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

[14] Manjeet Dahiya and Sorav Bansal. "Black-Box Equivalence Checking Across Compiler Optimizations". In: *Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings*. 2017, pp. 127–147. DOI: 10.1007/978-3-319-71237-6_7. URL: https://doi.org/10.1007/978-3-319-71237-6_7.

[15] Manjeet Dahiya and Sorav Bansal. "Modeling Undefined Behaviour Semantics for Checking Equivalence Across Compiler Optimizations". In: *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*. 2017, pp. 19–34. DOI: 10.1007/978-3-319-70389-3_2. URL: https://doi.org/10.1007/978-3-319-70389-3_2.

[16] Priyanka Darke et al. "VeriAbs: Verification by Abstraction and Test Generation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dirk Beyer and Marieke Huisman. Cham: Springer International Publishing, 2018, pp. 457–462. ISBN: 978-3-319-89963-3.

[17] Emanuele De Angelis et al. "Relational Verification Through Horn Clause Transformation". In: *Static Analysis*. Ed. by Xavier Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 147–169. ISBN: 978-3-662-53413-7.

[18] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[19] Saumya Debray, Robert Muth, and Matthew Weippert. "Alias Analysis of Executable Code". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: ACM, 1998, pp. 12–24. ISBN: 0-89791-979-3. DOI: 10.1145/268946.268948. URL: http://doi.acm.org/10.1145/268946.268948.

[20] *diet libc webpage*. https://www.fefe.de/dietlibc/. 2020.

[21] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[22] Niklas Een, Alan Mishchenko, and Robert Brayton. "Efficient Implementation of Property Directed Reachability". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, pp. 125–134. ISBN: 9780983567813.

[23] Michael D. Ernst et al. "The Daikon System for Dynamic Detection of Likely Invariants". In: *Sci. Comput. Program.* 69.1-3 (Dec. 2007), pp. 35–45. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.01.015. URL: http://dx.doi.org/10.1016/j.scico.2007.01.015.

[24] P. Ezudheen et al. "Horn-ICE Learning for Synthesizing Invariants and Contracts". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276501. URL: https://doi.org/10.1145/3276501.

[25] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. "Automated Discovery of Simulation Between Programs". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Martin Davis et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 606–621. ISBN: 978-3-662-48899-7.

[26] Dennis Felsing et al. "Automating Regression Verification". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 349–360. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642987. URL: http://doi.acm.org/10.1145/2642937.2642987.

[27] Cormac Flanagan and K. Rustan M. Leino. "Houdini, an Annotation Assistant for ESC/-Java". In: *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 500–517. ISBN: 3540417915.

[28] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Counterexample-Guided Correlation Algorithm for Translation Validation". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428289. URL: https://doi.org/10.1145/3428289.

[29] Shubhani Gupta et al. "Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition". In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Cham: Springer International Publishing, 2018, pp. 365–382. ISBN: 978-3-319-94144-8.

[30] Arie Gurfinkel et al. "The SeaHorn Verification Framework". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 343–361. ISBN: 978-3-319-21690-4.

[31] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[32] Hossein Hojjat et al. "A Verification Toolkit for Numerical Transition Systems". In: *FM 2012: Formal Methods*. Ed. by Dimitra Giannakopoulou and Dominique Méry. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 247–251. ISBN: 978-3-642-32759-9.

[33] Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. "Invariant Generation for Multi-Path Loops with Polynomial Assignments". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Cham: Springer International Publishing, 2018, pp. 226–246. ISBN: 978-3-319-73721-8.

[34] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[35] Theodoros Kasampalis et al. "Language-Parametric Compiler Validation with Application to LLVM". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 1004–1019. ISBN: 9781450383172.

[36] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. "Relational Program Reasoning Using Compiler IR". In: *J. Autom. Reason.* 60.3 (Mar. 2018), pp. 337–363. ISSN: 0168-7433. DOI: 10.1007/s10817-017-9433-5. URL: https://doi.org/10.1007/s10817-017-9433-5.

[37] Zachary Kincaid et al. “Non-linear Reasoning for Invariant Synthesis”. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 54:1–54:33. ISSN: 2475-1421. DOI: 10.1145/3158142. URL: http://doi.acm.org/10.1145/3158142.

[38] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. “Proving Optimizations Correct Using Parameterized Program Equivalence”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’09. Dublin, Ireland: ACM, 2009, pp. 327–337. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542513. URL: http://doi.acm.org/10.1145/1542476.1542513.

[39] Vaibhav Kiran Kurhe et al. “Automatic Generation of Debug Headers through BlackBox Equivalence Checking”. In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 144–154. DOI: 10.1109/CGO53902.2022.9741273.

[40] Shuvendu Lahiri et al. “Differential Assertion Checking”. In: *Foundations of Software Engineering (FSE’13)*. ACM, Aug. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=193772.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. “Compiler Validation via Equivalence Modulo Inputs”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 216–226. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594334. URL: http://doi.acm.org/10.1145/2594291.2594334.

[42] Xavier Leroy. “Formal certification of a compiler back-end, or: programming a compiler with a proof assistant”. In: *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. URL: http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf.

[43] Nuno P. Lopes et al. “Alive2: Bounded Translation Validation for LLVM”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 65–79. ISBN: 9781450383912. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[44] Nuno P. Lopes et al. "Provably Correct Peephole Optimizations with Alive". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, 2015, pp. 22–32. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737965. URL: http://doi.acm.org/10.1145/2737924.2737965.

[45] Saeed Maleki et al. "An Evaluation of Vectorizing Compilers". In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.68. URL: https://doi.org/10.1109/PACT.2011.68.

[46] Henry Massalin. "Superoptimizer: A Look at the Smallest Program". In: *ASPLOS '87: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 1987, pp. 122–126.

[47] David Menendez, Santosh Nagarakatte, and Aarti Gupta. "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM". In: Sept. 2016, pp. 317–337. ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7_16.

[48] Robin Milner. *An Algebraic Definition of Simulation Between Programs*. Tech. rep. Stanford, CA, USA, 1971.

[49] Dmitry Mordvinov and Grigory Fedyukovich. "Synchronizing Constrained Horn Clauses". In: *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Ed. by Thomas Eiter and David Sands. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 338–355. DOI: 10.29007/gr5c. URL: https://easychair.org/publications/paper/LlxW.

[50] Markus Müller-Olm and Helmut Seidl. "Analysis of Modular Arithmetic". In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 46–60. ISBN: 978-3-540-31987-0.

[51] KedarS. Namjoshi and LenoreD. Zuck. "Witnessing Program Transformations". English. In: *Static Analysis*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 304–323. ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_17. URL: http://dx.doi.org/10.1007/978-3-642-38856-9_17.

[52] George C. Necula. "Translation Validation for an Optimizing Compiler". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 83–94. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349314. URL: http://doi.acm.org/10.1145/349299.349314.

[53] ThanhVu Nguyen et al. "Using Dynamic Analysis to Discover Polynomial and Array Invariants". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 683–693. ISBN: 978-1-4673-1067-3. URL: http://dl.acm.org/citation.cfm?id=2337223.2337304.

[54] *OpenBSD libc sources*. https://github.com/openbsd/src/tree/master/lib/libc. 2020.

[55] Phitchaya Mangpo Phothilimthana et al. "Scaling Up Superoptimization". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 297–310. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872387. URL: http://doi.acm.org/10.1145/2872362.2872387.

[56] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '98. London, UK, UK: Springer-Verlag, 1998, pp. 151–166. ISBN: 3-540-64356-7. URL: http://dl.acm.org/citation.cfm?id=646482.691453.

[57] *Polybench/C*. https://sourceforge.net/projects/polybench/. 2019.

[58] Thomas Reps, Mooly Sagiv, and Greta Yorsh. "Symbolic Implementation of the Best Transformer". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernhard Steffen and Giorgio Levi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 252–266. ISBN: 978-3-540-24622-0.

[59] Davide Sangiorgi and Jan Rutten. *Advanced Topics in Bisimulation and Coinduction*. 1st. New York, NY, USA: Cambridge University Press, 2011. ISBN: 1107004977, 9781107004979.

[60] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 305–316. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451150. URL: http://doi.acm.org/10.1145/2451116.2451150.

[61] Rahul Sharma et al. "A Data Driven Approach for Algebraic Loop Invariants". In: *Proceedings of the 22Nd European Conference on Programming Languages and Systems*. ESOP'13. Rome, Italy: Springer-Verlag, 2013, pp. 574–592. ISBN: 978-3-642-37035-9. DOI: 10.1007/978-3-642-37036-6_31. URL: http://dx.doi.org/10.1007/978-3-642-37036-6_31.

[62] Rahul Sharma et al. "Conditionally Correct Superoptimization". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 147–162. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814278. URL: http://doi.acm.org/10.1145/2814270.2814278.

[63] Rahul Sharma et al. "Data-driven Equivalence Checking". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 391–406. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509509. URL: http://doi.acm.org/10.1145/2509136.2509509.

[64] Michael Stepp, Ross Tate, and Sorin Lerner. "Equality-based Translation Validator for LLVM". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 737–742. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032364.

[65] Ofer Strichman and Benny Godlin. "Regression Verification - A Practical Way to Verify Programs". English. In: *Verified Software: Theories, Tools, Experiments*. Ed. by Bertrand Meyer and Jim Woodcock. Vol. 4171. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 496–501. ISBN: 978-3-540-69147-1. DOI: 10.1007/978-3-540-69149-5_54. URL: http://dx.doi.org/10.1007/978-3-540-69149-5_54.

[66] Ross Tate, Michael Stepp, and Sorin Lerner. "Generating Compiler Optimizations from Proofs". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 389–402. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706345. URL: http://doi.acm.org/10.1145/1706299.1706345.

[67] Ross Tate et al. "Equality Saturation: a New Approach to Optimization". In: *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*. Savannah, GA, USA: ACM, 2009, pp. 264–276. ISBN: 978-1-60558-379-2. DOI: http://doi.acm.org/10.1145/1480881.1480915. URL: http://www.cs.cornell.edu/~ross/publications/eqsat/.

[68] A. Thakur et al. "PostHat and All That". In: *Electron. Notes Theor. Comput. Sci.* 311.C (Feb. 2015), pp. 15–32. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.02.003. URL: http://dx.doi.org/10.1016/j.entcs.2015.02.003.

[69] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating Value-graph Translation Validation for LLVM". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 295–305. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993533. URL: http://doi.acm.org/10.1145/1993498.1993533.

[70] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. "The Recognition of Series Parallel Digraphs". In: *SIAM Journal on Computing* 11.2 (1982), pp. 298–313. DOI: 10.1137/0211023. eprint: https://doi.org/10.1137/0211023. URL: https://doi.org/10.1137/0211023.

[71] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. "Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences". In: *ACM Trans. Program. Lang. Syst.* 34.3 (Nov. 2012). ISSN: 0164-0925. DOI: 10.1145/2362389.2362390. URL: https://doi.org/10.1145/2362389.2362390.

[72] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 978-1-4503-

0663-8. DOI: 10.1145/1993498.1993532. URL: http://doi.acm.org/10.1145/1993498.1993532.

[73] *Yices2 bug report*. https://github.com/SRI-CSL/yices2/issues/146. 2020.

[74] Anna Zaks and Amir Pnueli. "CoVaC: Compiler Validation by Program Analysis of the Cross-Product". In: *Proceedings of the 15th International Symposium on Formal Methods*. FM '08. Turku, Finland: Springer-Verlag, 2008, pp. 35–51. ISBN: 978-3-540-68235-6. DOI: 10.1007/978-3-540-68237-0_5. URL: http://dx.doi.org/10.1007/978-3-540-68237-0_5.

[75] Jianzhou Zhao et al. "Formal Verification of SSA-based Optimizations for LLVM". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 175–186. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462164. URL: http://doi.acm.org/10.1145/2491956.2462164.

[76] He Zhu, Stephen Magill, and Suresh Jagannathan. "A Data-driven CHC Solver". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 707–721. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192416. URL: http://doi.acm.org/10.1145/3192366.3192416.

[77] Lenore Zuck et al. "VOC: A Methodology for the Translation Validation of Optimizing Compilers". In: 9.3 (Mar. 28, 2003), pp. 223–247.

List of Publications

This thesis is based on the following publications

1. **Shubhani Gupta**, Aseem Saxena, Anmol Mahajan, Sorav Bansal. "*Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition*", in Theory and Applications of Satisfiability Testing (SAT) 2018. https://doi.org/10.1007/978-3-319-94144-8_22.
2. **Shubhani Gupta**, Abhishek Rose, and Sorav Bansal. "*Counterexample-guided correlation algorithm for translation validation*" in Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) 2020. https://doi.org/10.1145/3428289.

Other publications in this duration

1. Neetu Jindal, **Shubhani Gupta**, Divya Praneetha Ravipati, Preeti Ranjan Panda, Smruti R. Sarangi. "*Enhancing Network-on-Chip Performance by Reusing Trace Buffers*", in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 2020. https://doi.org/10.1109/TCAD.2019.2907909.

2. Vaibhav Kiran Kurhe, Pratik Karia, **Shubhani Gupta**, Abhishek Rose and Sorav Bansal. "*Automatic Generation of Debug Headers through BlackBox Equivalence Checking*", in International Symposium on Code Generation and Optimization (CGO) 2022. https://doi.org/10.1109/CGO53902.2022.9741273.

Biography

Shubhani is a PhD student of Amar Nath and Shashi Khosla School of Information Technology department of IIT Delhi. She is a B.Tech. gold medalist in Electronics and Communication Engineering from NIT Kurukshetra. Prior to joining the PhD program, Shubhani has worked with Bharat Electronics Limited (Ministry of Defence) for around five years. During her tenure, she played a key role in one of the pioneering technology being developed at BEL. She has also worked as Lead Engineer at Samsung R&D Institute Delhi for around two years and was awarded as the "Employee of The Quarter" for good contribution in project. Shubhani defended her PhD thesis in February 2023. Her research interests include: compiler verification and program analysis.

Shubhani secured an all India rank 29 in the National Talent Search Exam Scholarship awarded by Govt. of India. She was also awarded with "Kalpana Chawla Scholarship" for first AIEEE rank among girls in Haryana state. During her B. Tech., she was shortlisted for O. P. Jindal Scholarship twice and was declared as OPJEMS'08. Her PhD work has been selected for "Batch of 1969 Innovation Fellow Award" at IIT Delhi.