Ensuring Correctness of Compiled Code

by

Ganna Zaks

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May, 2009

Amir Pnueli

To Alex

Acknowledgments

This work would not be possible without the support and guidance of my advisor, Amir Pnueli. His deep knowledge, wisdom, and kindness are an infinite source of inspiration. Thank you for sharing them so generously with me along the way.

A part of this thesis is based on the research that I have done in collaboration with the Laboratory for Reliable Software (LaRS) at JPL. While working at LaRS, I had the opportunity to experience the research in the industry setting, received invaluable feedback on my work, and participated in many fun lunch time discussions. My deepest gratitude goes to Gerard Holzmann, Klaus Havelund, Alex Groce, and especially Rajeev Joshi who is my mentor and a great friend.

I would also like to thank my thesis committee members, Benjamin Goldberg, Clark Barrett, Patrick Cousot, and Radhia Cousot as well as the other members of the ACSys group for giving their feedback on early versions of my work and sharing their expertise. Many thanks to Christopher Conway who has volunteered to proofread my manuscripts and provide the valuable comments.

I thank Chris Lattner and the LLVM Compiler Infrastructure team for the creation a great open source product and their zealous commitment to support the LLVM users.

Last but not least, I am thankful to my family. To my parents, Tamila Surzhyk and Eduard Savustyanenko, for their encouragement and passion for knowledge that turned out to be contagious. To Alex, for his never-ending love, support, and the wonderful sense of humor. To my son Henry for the joy and happiness he gives me without even realizing it.

Abstract

Traditionally, the verification effort is applied to the abstract algorithmic descriptions of the underlining software. However, even well understood protocols such as Petersons protocol for mutual exclusion, whose algorithmic description takes only half a page, have published implementations that are erroneous. Furthermore, the semantics of the implementations can be altered by optimizing compilers, which are very large applications and, consequently, are bound to have bugs. Thus, it is highly desirable to ensure the correctness of the compiled code especially in safety critical and high-assurance software. This dissertation describes two alternative approaches that bring us closer to solving the problem.

First, we present Compiler Validation via Analysis of the Cross-Product (CoVaC) - a deductive framework for proving program equivalence and its application to automatic verification of transformations performed by optimizing compilers. To leverage the existing program analysis techniques, we reduce the equivalence checking problem to analysis of one system - a cross-product of the two input programs. We show how the approach can be effectively used for checking equivalence of single-threaded programs that are structurally similar. Unlike the existing frameworks, our approach accommodates absence of compiler annotations and handles most of the classical intraprocedural optimizations such as constant folding, reassociation, common subexpression elimination, code motion, branch optimizations, and others. In addition, we have developed rules for translation validation of interprocedural optimizations, which can be applied when compiler annotations are available.

The second contribution is the pancam framework for model checking multi-threaded C programs. pancam first compiles a multi-threaded C program into optimized bytecode format. The framework relies on Spin, an existing explicit state model checker, to orchestrate the program's state space search. However, the program transitions and states are computed by the pancam bytecode interpreter. A feature of our approach is that not only pancam checks the actual implementation, but it can also check the code after compiler optimizations. Pancam addresses the state space explosion problem by allowing users to define data abstraction functions and to constrain the number of allowed context switches. We also describe a partial order reduction method that reduces context switches using dynamic knowledge computed on-the-fly, while being sound for both safety and liveness properties.

Contents

Dedication **ii**

Acknowledgments **iii**

Abstract **v**

List of Figures **x**

List of Tables **xii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

2 Compiler Verification **7**

2.1 Background 8

2.2 Preliminaries 11

2.2.1 Transition Graphs 11

2.2.2 Inductive Assertion Network 17

2.2.3 The Notion of Correct Translation 19

2.2.4 Consonant Transition Graphs 22

2.2.5 Notation 25

2.3 Interprocedural Translation Validation 25

2.3.1 Interprocedural Translation Validation Algorithm 27

2.3.2 Invariants for Interprocedural Constant Propagation 29

2.3.3 Translation Verification Conditions 33

2.3.4 ITV Example: Constant Propagation and TRE 41

2.4 Compiler Validation via Analysis of the Cross-Product (CoVaC) . . . 44

2.4.1 Introduction 45

2.4.2 Comparison Graphs 47

2.4.3 Comparison Graph Construction 53

2.4.4 Correlating the Unbounded Heaps 63

2.4.5 CoVaC Implementation 69

2.4.6 Experimental Results 75

2.5 Related Work 76

2.6 Conclusions 79

3 Verifying multithreaded C programs with pancam 81

3.1 Introduction 81

3.2 Background 83

3.2.1 Spin 83

3.2.2 Model Driven Verification 86

3.3 Model Checking C programs with pancam 87

3.4 Addressing State Space Explosion 93

3.4.1 Abstraction 94

3.4.2 Context-Bounded Checking 96

3.5 On-the-fly Dynamic Partial Order Reduction 97

3.5.1 Preliminaries 99

3.5.2 Superstep Partial Order Reduction 102

3.5.3 Application to Multithreaded Programs 106

3.5.4 Implementation of Superstep Reduction in pancam 108

3.6 Experimental Results 114

3.7 Related Work 118

4 Conclusion **121**

A CoVaC Proofs **124**

B Correctness of SPOR **129**

Bibliography **137**

List of Figures

2.1 The procedures 11

2.2 Transition graphs for the program that given A, outputs $(3 * A)! + 5$. 14

2.3 Call and return transitions 16

2.4 Call verification conditions for inductive assertion network 18

2.5 Inductive network for interprocedural constant copy propagation . . . 32

2.6 Call verification conditions 35

2.7 Inlining verification conditions 37

2.8 Tail recursion elimination verification conditions 39

2.9 ITV example: the program before and after compilation 42

2.10 A comparison transition graph for $f(\&(Y, y))$ 51

2.11 Algorithm `compose` that constructs $f = f^S \boxtimes f^T$ 55

2.12 Misaligned branches 57

2.13 CoVaC example: the source, the target, and the comparison graphs 61

2.14 Heaps equivalence example 63

2.15 The work flow of the CoVaC tool 70

2.16 Dependency between the running time and the size of the procedure . 75

3.1 Interaction of Spin and pancam 88

3.2 Excerpt of Peterson's Algorithm from the Wikipedia 89

3.3 LLVM bytecode for function `pa_desc_lock` 90

3.4 Spin driver for executing pancam on Peterson's Algorithm 92

3.5 How state is maintained by pancam 94

3.6 Code for implementing context bounding with pancam 96

3.7 Correctness of SPOR 105

3.8 Application of SPOR to threads 108

3.9 Pseudocode of `pan_step` with superstep reduction. 110

3.10 Pseudocode of the independence condition tester 111

3.11 The example of the Superstep POR algorithm 113

3.12 Context bounding for peterson.c with bug 115

3.13 Context bounding for peterson.c without bug 116

3.14 The Robot example 117

List of Tables

2.1 The heap delta sets 66

3.1 Other experiments with pancam 117

Chapter 1

Introduction

1.1 Motivation

Computer technology plays such an important role in our lives that we cannot imagine living without it. We have learned to depend on it to store our private information, provide the means of communication, and assist with everyday activities such as banking and shopping. Moreover, we entrust the software systems with our finances, health care, and our lives (at least those of us who fly on airplanes or drive cars). In many cases, the new technologies allow achievements that could never be possible otherwise. However, there is an implicit assumption that the automated systems are going to exhibit the desired behavior. The truth is that not only this is not always the case but it is often even hard to precisely state what the desired behavior is.

Since we rely on computers to make critical decisions, the software bugs can have extremely serious consequences. Here are several examples ranging from financial and asset loss to threatening human lives.

- In the late eighties, several people had died after being overdosed by the Therac-25 radiation therapy machines. It is believed that the problem was caused by malfunctioning software [46].
- More recently in 2008, a team of computer science researchers found security vulnerabilities in a heart defibrillator and a pacemaker, which allowed them to reprogram the device to shut down or deliver jolts of electivity that would potentially be fatal [30].
- The North America blackout, which effected an estimated 50 million people and caused at least eleven fatalities, was triggered by a local outage that went undetected due to a race condition in the General Electric Energy's monitoring software. The bug prevented alarms from showing on the control system [41].

Software verification and program analysis provide systematic approaches to ensuring that a program satisfies its specification. However, a key challenge in applying the formal methods tools to software is the difficulty of verifying properties of implementation code, as opposed to checking abstract algorithmic descriptions. Even well understood protocols such as Petersons protocol for mutual exclusion, whose algorithmic description takes half a page, have published implementations that are erroneous. This is especially a problem for programs written in the C language, which has several features (such as function pointers, pointer arithmetic and arbitrary type casting, to name a few) that are difficult to model faithfully. It is an unfortunate fact of life that the programs most in need of verification are those that use such constructs the most, viz., C programs for embedded systems.

However, checking the C implementation might not be enough. The ultimate goal is to ensure correctness of the executable. This brings us to verification of optimizing compilers. Compilers are quite large applications, which are bound to have bugs. For example, the GCC Bug Database contains over 3 thousand reported defects some of which may alter the behavior of a program being compiled. This is highly undesirable, especially in safety critical and high-assurance software.

1.2 Contributions

This dissertation describes two orthogonal approaches that bring us closer to solving the problem stated above. The first approach is verification of optimizing compilers. The second one is model checking of the optimized bytecode.

Chapter 2 presents two novel deductive verification frameworks, which are focused on the optimization phase of the compiler. They are based on the idea of translation validation [53]. Translation validation is an automatic approach of ensuring compilation correctness in which each compiler run is followed by a validation pass that proves that the target code produced by the compiler is a correct implementation (translation) of the source code. The approaches assume that the same intermediate representation is used to specify both input systems. In addition, to make the methodology effective, we restrict our attention to consonant (or structurally similar) programs in which each loop of the source has a corresponding loop in the target. The assumption is that one has to rely on the other tools to check the correctness of the implementation as well as the validity of the other compiler phases (such as code generation and parsing).

Interprocedural Translation Validation (ITV), presented in Section 2.3, is an extension of the existing translation validation framework TVOC [21, 67, 68, 39] to procedural reactive programs. In addition to the structure preserving optimizations handled by TVOC, it accommodates most classical interprocedural optimizations such as global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning.

Section 2.4 presents Compiler Validation via Analysis of the Cross-Product (CoVaC). The main distinction that sets CoVaC apart is that the translation validation problem is reduced to checking properties of a single program – a cross-product of the two input programs. This allows us to effortlessly leverage the existing program analysis techniques and tools. We show how the approach can be effectively used for checking equivalence of consonant (i.e., structurally similar) programs and report on the prototype tool that applies the developed methodology to verification of LLVM compiler [7]. Unlike the other existing frameworks, CoVaC accommodates absence of compiler annotations and handles most of the classical intraprocedural optimizations such as constant folding, reassociation, common subexpression elimination, code motion, dead code elimination, branch optimizations, and others.

Invariant generation algorithms serve as a back bone of the compiler validation tools. We have developed two novel algorithms. The first one is presented in Section 2.3.2. It constructs a network of invariants necessary to prove the correctness of compilation when the context-sensitive constant copy propagation optimizes the source program. Second, Section 2.4.4 presents data flow analysis for determining the relation between the heaps of the source and the target programs. The invariants implied by the analysis are required for checking equivalence of the code with dynamically allocated data structures.

Chapter 3 presents an approach orthogonal to compiler verification. It describes pancam – a framework for model checking multi-threaded C programs. The framework allows to direct the power of the Spin model checker [8] to verification of optimized bytecode. In our framework, a multi-threaded C program is compiled into a typed bytecode format. pancam uses the internal virtual machine that interprets the bytecode and computes new program states under the direction of the Spin model checker. pancam combats the state space explosion problem by allowing users to define data abstraction functions and to constrain the number of allowed context switches. We also describe a partial order reduction method that reduces context switches using dynamic knowledge computed on-the-fly, while being sound for both safety and liveness properties.

Thus, the contribution of this thesis can be summarized under the following headings:

- An algorithm for interprocedural translation validation (ITV).
- The framework for compiler verification via program analysis of the cross-product (CoVaC).
- A methodology for model checking multithreaded bytecode (pancam).

In the following two chapters, we elaborate on each of these results. It might seem that some of these frameworks are redundant. For example, one might ask: Why verify the compiler when we can model check the bytecode? Chapter 4 addresses these concerns. It shows that each approach has its weaknesses and strengths, which define the settings where one would be preferable to the other.

Chapter 2

Compiler Verification

Compilers, especially optimizing compilers, are quite large applications, which are bound to have bugs. For example, the GCC Bug Database contains over 3 thousand reported bugs some of which may alter the behavior of programs being compiled. This is highly undesirable, especially in safety critical and high-assurance software, where the effort of program correctness verification is extensive. First, the developers manually examine code and test it. Then, numerous verification tools and techniques are applied to verify that the source code satisfies the desired properties. After all the rigorous checks are complete, the program is compiled by an optimizing compiler and released. Clearly, the verification effort should not stop here – it is highly advisable to ensure that the transformations performed by a compiler preserve the semantics of a program.

2.1 Background

The goal of a compiler verification framework is to check that the source and the target programs are observationally equivalent assuming that the source program has well-defined semantics. This assumption comes from the fact that the compilers are free to interpret the code that is not well-defined. For example, a C compiler could evaluate the expression $(x * y * z)$ by first evaluating z, then x, then y, and then multiplying. Note that if the operands are not side effect free, the evaluation order may influence the observable behavior of the program. In these situations, the verifier may or may not raise the error. This comes from the fact that, like compiler, the verifier is free to interpret the under-specified constructs. Even though it is desirable to catch errors that exploit the weaknesses of the language semantics, it is not always possible. The best approach, which we follow in this work, is to interpret the under-specified constructs similarly to the majority of the compilers (or if dealing with one particular compiler, in the same way as the compiler in question).

The methodology of compiler verification can be categorized by its intended customers (i.e., by the ultimate consumer). *Compiler writers* may assume full knowledge of the inner workings of a particular compiler and are interested in methods that lead to creation of a self-certified compiler. Given a source program, *a certified compiler* either produces a target program observationally equivalent to the source or raises an error. An impressive effort in this direction is presented in [45, 14], which describes a formal certification of a a complete compilation chain using the Coq proof assistant. Another approach is taken in [44, 43], which present languages for specification of compiler optimizations that can be automatically proved sound, meaning that their transformations are always semantics-preserving. However, the latter approach assumes the correctness of the execution engine and focuses only on the optimization phase of the compiler.

Another group interested in compiler verification are *compiler users* who may need to work with a black box and require tools that accommodate minimal compiler cooperation. Good examples of such tools are presented in [51], [59], and [67]. The three tools are based on the technique of translation validation even though the implementations are quite different – [51] and [59] are based on symbolic evaluation, whereas [67] generates verification conditions and checks their validity with an automatic theorem prover.

First introduced in [53], *translation validation* ensures that optimizing transformations preserve program semantics. In essence, instead of attempting the verification of a given compiler, each compiler run is followed by a validation pass that automatically checks that the target code produced by the compiler is semantically equivalent to the source code. A good question is: "Can this goal be achieved?" The problem of program equivalence is undecidable. However, since the focus is only on compiler optimizations, the number of false alarms can be drastically minimized or even eliminated, intuitively, due to the fact that we are aware of the analyses used by the optimizing compilers, and since those analyses are mechanical in nature.

In a setting of a certified compiler, translation validation can be applied to verify correctness of particular optimization passes. For example, [63] and [62] present translation validation algorithms for lazy code motion and instruction scheduling optimizations along with mechanical proofs of the algorithms' correctness. Whereas [40] applies translation validation to verification of register allocation. These methods choose relative completeness (with respect to the property being checked) and efficiency over generality.

Most of the general compiler verification frameworks [51, 59, 67, 58] are based on translation validation. Even though the definition only mentions two programs (the source and the target) as the input, in practice, the algorithms rely on heuristics and compiler debug information to suggest the correspondence between the variables and locations of the source and target programs. Credible compilation [58] carries this dependency to the extreme: the compiler is responsible for annotating the source code with a full proof so that translation validation reduces to proof checking. In Section 2.4, we present CoVaC framework. Even though CoVaC can be used for construction of certifying compilers, we also show how it can be used in case of a black-box compiler, where no annotations are available. As far as we know, this is the first work in this direction.

The existing translation validation approaches are not capable of, and were not designed to deal with, interprocedural optimizations. For example, in [51] two executions are considered the same if both lead to the same sequence of function calls and returns. In Section 2.3, we present a framework for translation validation of compiler optimization run that targets *reactive procedural* programs. The algorithm extends [67] to accommodate most classical *interprocedural* optimizations such as global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning.

2.2 Preliminaries

This section starts with a description of the transition graphs, which are used to model the procedural reactive programs with intermediate input and output instructions. Next, we describe the inductive assertion network - a set of program assertions satisfying the set of the corresponding verification conditions. Such assertions are program invariants. In addition, the set of the verification conditions constitutes an inductive proof of this fact. Our model and the definition of the assertion network are similar to those presented in [55]. Section 2.2.3 presents the notion of correct translation used in this work to show that two input programs are equivalent. Finally, we define the term "consonant programs" in Section 2.2.4.

2.2.1 Transition Graphs

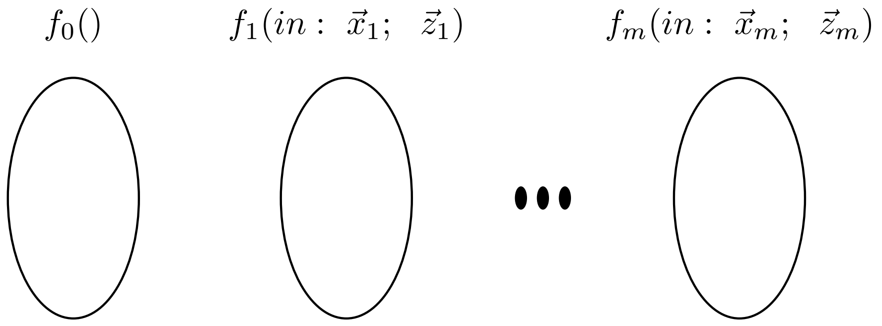

$f_0()$
$f_1(in:\ \vec{x}_1;\ \ \vec{z}_1)$
$f_m(in:\ \vec{x}_m;\ \ \vec{z}_m)$

Figure 2.1: The procedures

A program (application) $\mathcal{S}$ consists of $m+1$ procedures: *main*, $f_1,\ \ldots,\ f_m$, where *main* represents the main procedure, and $f_1,\ \ldots,\ f_m$ are procedures which may be called from *main* or from other procedures. Each procedure f is represented by a *transition graph* $f := (\vec{y},\ \mathcal{N},\ \mathcal{E})$ with variables $\vec{y}$, a set of nodes (locations)

$\mathcal{N}$ and a set of labeled edges $\mathcal{E}$. A program $\mathcal{S}$ is modeled by a forest that includes all the transition graphs of its $m+1$ procedures. We use the subscript notation to differentiate between the nodes, variables, and edges of different procedures when the association is not clear from the context.

We use $\vec{y}$ to denote the typed variables of a procedure; $\vec{y} = (\vec{x};\ \vec{z};\ \vec{w})$, i.e. the variables in $\vec{y}$ are partitioned into $\vec{x}$, $\vec{z}$, and $\vec{w}$, where $\vec{x}$ and $\vec{z}$ are the *input* parameters and $\vec{w}$ denotes the *local* variables. We use $f(\vec{x};\ \&\vec{z})$ to denote the signature of a procedure. Here, call-by-value parameter passing method is used for $\vec{x}$, and call-by-reference is used for $\vec{z}$. A procedure may return a result by means of $\vec{z}$ variables. To simplify the presentation, we assume that the *main* procedure does not have any arguments. The variables range over one of the following domains: *int*, *real*, or *map*. A variable of type *map* is a partial heterogeneous mapping from addresses of type *int* to values of *int* or *real* type. For a variable H of type *map*, expression $H[addr]$ denotes the value stored at address *addr*.

Each transition graph must have a distinct root node r as its only entry point, a distinct tail node t as its only exit point, and every other node must be on a path from r to t. Nodes of the graph are connected by directed edges labeled by instructions. There are four types of instructions: guarded assignments, procedure calls, reads, and writes. Consider a procedure $f(\vec{x};\ \&\vec{z})$ with $\vec{y} = (\vec{x},\ \vec{z},\ \vec{w})$. Let $\vec{u}$ include variables from $\vec{y}$; and $E(\vec{y})$ be a list of expressions over $\vec{y}$.

- A *guarded assignment* is an instruction of the form $c \rightarrow [\vec{u} := E(\vec{y})]$, where guard c is a boolean expression. When the assignment part is empty, we abbreviate the label to a pure condition c?.

- *Procedure call* instruction $g(E(\vec{y}), \vec{u})$ denotes a call to procedure $g(\vec{x}_g;\ \&\vec{z}_g)$, passing input parameters $E(\vec{y})$ by value and $\vec{u}$ by reference.

- *Read* and *write* instructions are denoted by $read(\vec{u})$ and $write(\vec{u})$. They are used to express the interaction of the procedure with the outside world; e.g. I/O instructions, reads and writes of C volatile variables, API calls.

The implicit guards of read, write, and procedure call instructions always evaluate to *true*. A transition graph is *deterministic* when, for every node n, the guards of all edges departing from n are mutually exclusive. A transition graph is *non-blocking* when, for every node, the disjunction of the guards evaluates to *true*. In this work, we only consider deterministic non-blocking systems.

Transition graphs can be used to model programs written in procedural languages. In order to construct a formal model of a program, we first choose a set of program locations Υ such that:

- At least one location in each loop belongs to Υ.

- For every procedure, both procedure entry and exit belong to Υ.

- The locations immediately before and after reads, writes, and procedure calls belong to Υ.

Each procedure whose implementation is given is represented by a transition graph. We choose the set Υ of a procedure f to be the set of nodes for the corresponding transition graph. For every pair of locations $n,\ m$ in Υ, if there exists a path π from n to m, which does not pass through any other location from Υ, we add edge (n, m)

to the graph and label it by the instruction that summarizes the effect of executing the path π. In general, the choice of Υ can be generalized not to require at least one location per each loop as long as we can ensure that the transitions between every pair of locations are computable [39]. Each call to a procedure whose implementation is hidden can be modeled by read/write instructions. If a hidden procedure is stateless and does not perform I/O operations (for example, *pow* function in C), the call is modeled by uninterpreted functions. Note that global variables and functions also can be modeled in this framework.

Example 1. *The procedure MAIN depicted on Fig. 2.2 reads in a natural number A and writes out the expression $(3 * A)! + 5$. It calls a recursive procedure FACT to compute the factorial. In FACT, the argument N is passed by value and R is passed by reference. FACT computes $N! * R$ and returns it to the caller by reference.*

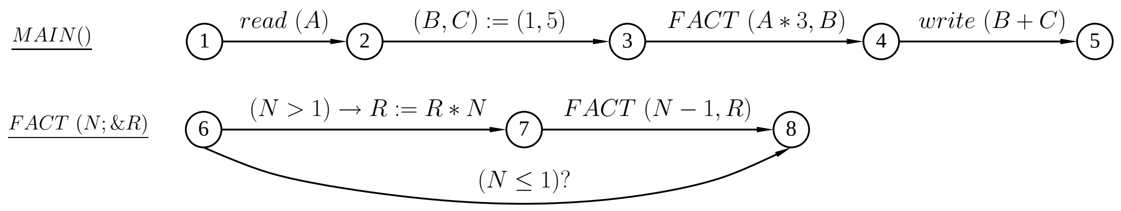


Figure 2.2: Transition graphs for the program that given A, outputs $(3 * A)! + 5$

States and Computations

We denote by $\vec{d} = (\vec{\xi};\ \vec{\zeta};\ \vec{\eta})$ a tuple of values, which represents an interpretation (i.e., an assignment of values) of the procedure variables $\vec{y} = (\vec{x};\ \vec{z};\ \vec{w})$.

We denote by $\vec{d} = (\vec{d^x};\ \vec{d^z};\ \vec{d^w})$ a tuple of values, which represents an interpretation (i.e., an assignment of values) of the procedure variables $\vec{y} = (\vec{x};\ \vec{z};\ \vec{w})$. A *state* of a procedure f is a pair $\langle n; \vec{d}\rangle$ consisting of a node n and a data interpretation $\vec{d}$. A $(\vec{\xi}, \vec{\zeta})$-*computation* of procedure f is a maximal sequence of states and labeled transitions:

$$\sigma : \langle r;\ (\vec{\xi}, \vec{\zeta}, \vec{\top})\rangle\ \xrightarrow{\lambda_1}\ \langle n_1;\ \vec{d_1}\rangle\ \xrightarrow{\lambda_2}\ \langle n_2;\ \vec{d_2}\rangle\ \ldots$$

The tuple $\vec{\top}$ denotes uninitialized values. At the first state of the computation, the location is r, the entry location of f; the values of input variables $\vec{x}$ and $\vec{z}$ are set to $\vec{\xi}$ and $\vec{\zeta}$, respectively, and the local variables $\vec{w}$ are not initialized. Labels of the transitions are either labels of edges in the program or the special return label. Each transition in a computation must be justified by one of the following cases:

- Intra-procedural transition $\langle l;\ \vec{d}\rangle\ \xrightarrow{op}\ \langle l';\ \vec{d'}\rangle$:
  - *Guarded Assignment*: There exists an edge e from node l to node l' in the program $\mathcal{S}$ (not necessarily in f) with label *op* that equals to $c \rightarrow [\vec{u} := E(\vec{y})]$ such that $\vec{d} \models c$ and $\vec{d'}\ =\ (\vec{d}\ with\ \vec{u} = E(\vec{d}))$, i.e. $\vec{d'}$ is obtained from $\vec{d}$ by replacing the values corresponding to the variables $\vec{u}$ by $E(\vec{d})$.
  - *Read*: There exists an edge e in the program $\mathcal{S}$ from node l to node l' with label *op* that equals to $read(\vec{u})$ such that $\vec{d'_v}\ =\ \vec{d_v}$, where $\vec{v}$ are the variables of $\vec{y}$ which are not in $\vec{u}$; and $\vec{d_v}$ is obtained from $\vec{d}$ by restricting it only to the values that correspond to the variables $\vec{v}$. The values of all variables but the ones in $\vec{u}$ are preserved by the read transition.
  - *Write*: There exists an edge e in the program $\mathcal{S}$ from node l to node l' with label *op* that equals to $write(\vec{u})$. Since write instruction does not change the values of the variables, $\vec{d'}\ =\ \vec{d}$.

- *Procedure call*: Consider the case when procedure f calls procedure g as depicted in Fig. 2.3. To justify transition $\langle l;\ \vec{d}\rangle \xrightarrow{g(E(\vec{y}),\vec{u})} \langle r_g;\ (E(\vec{d}), \vec{d_u}, \vec{\top})\rangle$, there must exist a call edge $e = (l, l')$ in the program $\mathcal{S}$ labeled by $g(E(\vec{y}), \vec{u})$. The location of the new state r_g is the first location in the called procedure g. $E(\vec{d})$ and $\vec{d_u}$ are the values of the input variables $\vec{x}_g$ and $\vec{z}_g$ on entry to g. We assume that the working variables are uninitialized.

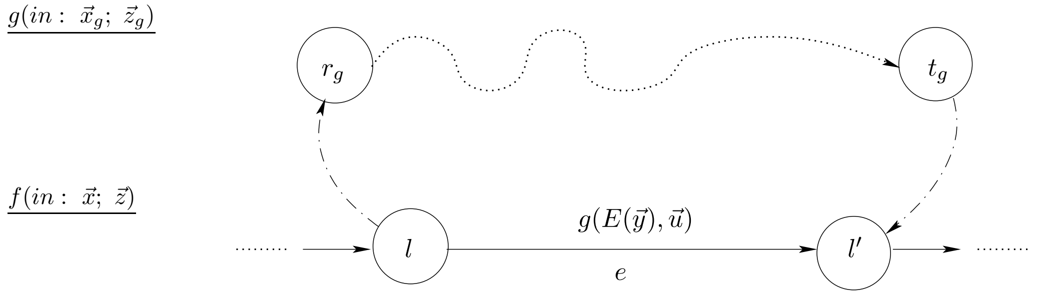


Figure 2.3: Call and return transitions

- *Procedure return*: Finally we consider transition $\langle t_g;\ (\xi'_g, \zeta'_g, \eta_g)\rangle \xrightarrow{ret_g} \langle l';\ \vec{d'}\rangle$. To justify such a transition, there must exist a procedure g (the procedure from which we return), such that t_g is the terminal location of g, and we should be able to identify a suffix of the current computation of the form

$$\langle l; \vec{d}\rangle \xrightarrow{g(E(\vec{y}),\vec{u})} \underbrace{\langle r_g; (\xi_g, \zeta_g, \vec{\top})\rangle \xrightarrow{e_1} \ldots \xrightarrow{e_m} \langle t_g; (\xi'_g, \zeta'_g, \eta_g)\rangle}_{\hat{\sigma}} \xrightarrow{ret_g} \langle l'; \vec{d'}\rangle$$

such that the segment $\hat{\sigma}$ is *balanced* (has an equal number of calls and returns). We also require that there is a procedure call edge from node l to node l' labeled by $g(E(\vec{y}), \vec{u})$ and $\vec{d'} = (\vec{d}\ with\ \vec{u} = \zeta'_g)$.

We use $Cmp(f)$ to denote the computations of a transition graph f. We define a set of computations of a procedural program $\mathcal{S}$, denoted $Cmp(\mathcal{S})$, to be the set of computations $Cmp(main)$.

2.2.2 Inductive Assertion Network

We introduce *virtual* variables $\vec{X}$ and $\vec{Z}$ to represent the values of the input variables $\vec{x}$ and $\vec{z}$ at the procedure entry and denote the extended vector of variables by $\vec{Y} = (\vec{X},\ \vec{Z},\ \vec{x},\ \vec{z},\ \vec{w})$. An **assertion network** associates an assertion φ_l with each program location l.

- For each procedure f with the entry location r, we denote φ_r by p_f. The **input predicate** $p_f = p_f(\vec{X}, \vec{Z};\ \vec{x}, \vec{z})$ imposes constraints only on the input variables of the procedure. Since we assume that the main procedure *main* does not have input parameters, $p_0 = true$.
- Similarly, we denote φ_t, the assertion associated with the exit location t of f, by q_f. The **output predicate** $q_f = q_f(\vec{X}, \vec{Z};\ \vec{z})$ is the procedure summary: it specifies the relation between the input and output values.
- The assertions at all other locations $\varphi_l(\vec{Y})$ may depend on any of the variables.

For each edge of the transition graph e connecting a node i to a node j, we form **verification conditions**, which represent different edge types:

- *Guarded Assignment*: If e is an assignment edge labeled by $c \rightarrow [\vec{u} := E(\vec{y})]$,

  $$\mathcal{VC}_e : \quad \varphi_i(\vec{Y}) \wedge c(\vec{y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})],$$

  where $\varphi_j(\vec{Y})[\vec{u} \mapsto E(\vec{y})]$ is obtained from $\varphi_j(\vec{Y})$ by replacing variables in $\vec{u}$ by the corresponding expressions in $E(\vec{y})$.

- *Read*: If e is a read edge labeled by $read(\vec{u})$,

$$\mathcal{VC}_e : \quad \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{u'}],$$

where $\vec{u'}$ is a vector of fresh variables. Intuitively, the assertion φ_j must hold for all possible inputs.

- *Write*: If e is a write edge labeled by $write(\vec{u})$,

$$\mathcal{VC}_e : \quad \varphi_i(\vec{Y}) \rightarrow \varphi_j(\vec{Y}).$$

- *Procedure call*: Last, consider the case when procedure f calls procedure g as depicted in Fig. 2.4.

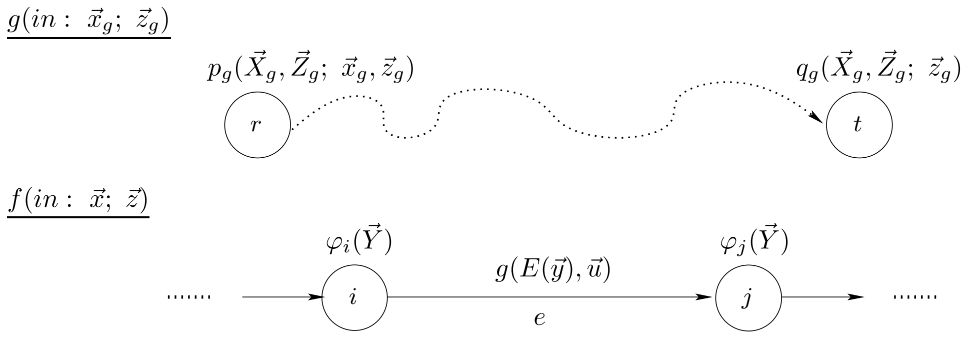

$g(in: \vec{x}_g;\ \vec{z}_g)$
$p_g(\vec{X}_g, \vec{Z}_g;\ \vec{x}_g, \vec{z}_g)$
$q_g(\vec{X}_g, \vec{Z}_g;\ \vec{z}_g)$
r
t
$f(in: \vec{x};\ \vec{z})$
$\varphi_i(\vec{Y})$
$\varphi_j(\vec{Y})$
$g(E(\vec{y}), \vec{u})$
i
j
e

Figure 2.4: Call verification conditions for inductive assertion network

We associate the following two conditions with a procedure call $g(E(\vec{y}), \vec{u})$, which calls the procedure with signature $g(\vec{x}_g;\ \&\vec{z}_g)$:

$$\mathcal{VC}_{call} : \quad \varphi_i(\vec{Y}) \rightarrow p_g(E(\vec{y}), \vec{u};\ E(\vec{y}), \vec{u})$$

$$\mathcal{VC}_{return} : \quad \varphi_i(\vec{Y}) \wedge q_g(E(\vec{y}), \vec{u};\ \vec{z}_g) \rightarrow \varphi_j(\vec{Y})[\vec{u} \mapsto \vec{z}_g]$$

Note that p_g and q_g are the input and output predicates of g. Thus, $\mathcal{VC}_{call}$ checks that the assertion associated with the location before the call, φ_i, implies the input predicate of the callee. $\mathcal{VC}_{return}$ checks that the assertion at the location reached immediately after the procedure return is implied by the output predicate and φ_i. The conditions generally use variables of the caller procedure with the only exception of the variables passed by reference $\vec{z}_g$. This exception allows us to disregard the old information about the variables passed by reference, stored by $\varphi_i(\vec{Y})$, and instead rely on the new information stored in q_g.

An assertion network $\Phi = \{\varphi_0, \ldots, \varphi_n\}$ for a program $\mathcal{S}$ is said to be **inductive** if all the verification conditions for all edges in $\mathcal{S}$ are valid. Network Φ is said to be **invariant** if for every execution state $\langle l; \vec{d} \rangle$ occurring in a computation, the visiting data state d satisfies the corresponding assertion φ_l associated with l.

Claim 1. *Every inductive network is invariant.*

2.2.3 The Notion of Correct Translation

We define the correctness of translation via equivalence of program behaviors that can be observed by the user. Intuitively, given the same input, both the source program $\mathcal{S}$ and the target program $\mathcal{T}$ must produce the same output and should either both terminate or generate infinite computations.

Given a computation, we define V_s, the set of *observable variables* at a state $s = \langle n; d \rangle$, to be the minimal set satisfying the following conditions:

- If s is a state immediately after transition $read(\vec{u})$, $V_s \supseteq \vec{u}$.
- If s is a state immediately before transition $write(\vec{u})$, $V_s \supseteq \vec{u}$.

Above, we use $V_s \supseteq \vec{u}$ to denote $V_s \supseteq \{v : v \text{ in } \vec{u}\}$.

We associate *observation function* $\mathcal{O}$ with each program, mapping the states and transition labels of the source and target programs into a common domain. The observation function needs to ensure that the observable states and transitions of the source and target computations match. Formally, given a state $s = \langle n; d \rangle$, an observation function $\mathcal{O}(s)$ is defined as following. Let V_s be the set of observable variables at s. If $V_s = \emptyset$ then $\mathcal{O}(s) = \perp$, else $\mathcal{O}(s) = \vec{d}_{V_s}$. We obtain $\vec{d}_{V_s}$ by restricting $\vec{d}$ only to the values that correspond to the variables in V_s. Given a transition label λ, an observation function $\mathcal{O}(\lambda)$ is defined as follows. If λ is a label of a transition that is a read, $\mathcal{O}(\lambda) = \mathit{read}$. If λ is a label of a transition that is a write, $\mathcal{O}(\lambda) = \mathit{write}$. Otherwise, $\mathcal{O}(\lambda) = \perp$.

An *observation* of a computation σ, denoted $o(\sigma)$, is obtained by applying the observation function $\mathcal{O}$ to each state and each transition label in σ. That is, for

$\sigma : s_1 \xrightarrow{\lambda_1} s_2 \xrightarrow{\lambda_2} s_3 \ldots ,$

we get

$o(\sigma) : \mathcal{O}(s_1) \xrightarrow{\mathcal{O}(\lambda_1)} \mathcal{O}(s_2) \xrightarrow{\mathcal{O}(\lambda_2)} \mathcal{O}(s_3) \ldots .$

Definition 1. *Computations σ and σ' are* **stuttering equivalent***, denoted $\sigma \sim_{st} \sigma'$, if their observations $o(\sigma)$, $o(\sigma')$ only differ from each other by finite sequences of pairs $\perp \xrightarrow{\perp}$ or $\xrightarrow{\perp} \perp$.*

Stuttering equivalence is used to ensure that even though the programs may have to execute a different number of instructions to get to an observable state, the difference is always finite. Our assumption is that the user is not time-sensitive so this finite delta cannot be observed. For example, $\beta \sim_{st} \beta'$:

$$o(\beta):\ \perp \xrightarrow{read} (5,22) \xrightarrow{\perp} \perp \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp$$
$$o(\beta'):\ \perp \xrightarrow{read} (5,22) \xrightarrow{\perp} \perp \xrightarrow{\perp} (110) \xrightarrow{write} \perp \xrightarrow{\perp} \perp$$

In both computations, first two numbers: 5 and 22, are read; and then, after a finite number of steps, their product: 110, is written out.

Definition 2. *We say that procedure $f^{\mathcal{T}}$ is* **a correct translation** *of procedure $f^{\mathcal{S}}$ if, for every $(\vec{\xi}, \vec{\zeta})$-computation $\sigma^{\mathcal{T}}$ in $Cmp(f^{\mathcal{T}})$, there exists a $(\vec{\xi}, \vec{\zeta})$-computation $\sigma^{\mathcal{S}}$ in $Cmp(f^{\mathcal{S}})$ such that $\sigma^{\mathcal{T}} \sim_{st} \sigma^{\mathcal{S}}$, and vice versa. Program $\mathcal{T}$ is a correct translation of program $\mathcal{S}$ if $main^{\mathcal{T}}$ is a correct translation of $main^{\mathcal{S}}$.*

Notice, that this definition of correct translation defines an *equivalence relation*. This is why, later in this work, we may say that two programs (or procedures) are equivalent whenever one program (or procedure) is a correct translation of another.

The definition above is the most general definition of an observation, which ensures that all the program behavior visible to the outside world is being accounted for. However, in some cases we have to strengthen this definition. In particular, when we do not perform whole program analysis but rely solely on intraprocedural reasoning, we have to observe the input and output parameters at the time of a procedure call as well as at the time of a procedure return. The stronger definition allows us to apply an assume-guarantee reasoning when proving the correctness of translation, where procedures f^S and f^T are proved to be equivalent, assuming that all the corresponding callees are equivalent themselves. That is the case in the CoVaC framework, which is presented in Section 2.4.

In order to enforce the additional tracking, we extend the definitions of the observable variables and the observation function as follows. We define V_s, the set of the variables observable at a state s, to be the minimal set satisfying the following four conditions (two of which are the same as in the previous definition):

- If s is a state immediately after transition $read(\vec{u})$, $V_s \supseteq \vec{u}$.
- If s is a state immediately before transition $write(\vec{u})$, $V_s \supseteq \vec{u}$.
- If $n = r$ is the entry node of procedure $f(\vec{x},\ \&\vec{z})$, $(V_s \supseteq \vec{x}) \wedge (V_s \supseteq \vec{z})$.
- If $n = t$ is the exit node of procedure $f(\vec{x},\ \&\vec{z})$, $V_s \supseteq \vec{z}$.

The observation function applied to a state $\mathcal{O}(s)$ is defined as before. However, in addition to observing a transition label that is a read or a write, we should also observe a procedure call and return labels. Let g be a name of some procedure. If λ is a label of a transition that is a call to the procedure g, or a return from g, $\mathcal{O}(\lambda)$ is equal to $call_g$ or ret_g, respectively.

2.2.4 Consonant Transition Graphs

In this section, we give a definition of consonant or structurally similar programs. This is an attempt to formalize the informal notion of structure preserving transformations used in the literature [67].

We are going to use the source program $\mathcal{S}$ with the set of nodes $\mathcal{N}^S$ and the set of edges $\mathcal{E}^S$ to define several notions, which apply to both the source program $\mathcal{S}$ and the target program $\mathcal{T}$. Each node of $\mathcal{S}$ belongs to one of the following categories: read, write, call, branch, unconditional assignment, or exit; denoted *rd*, *wt*, *cl*, *br*, *ua*, and *tl* respectively. Intuitively, the type of a node n depends on the type of the edges outgoing from n. Specifically, we say that a node $n \in \mathcal{N}^S$ is a read node, written $\tau(n^S) = rd$, if $\exists\ (n^S, m^S) \in \mathcal{E}^S$ and (n^S, m^S) is labeled by a read instruction. Similarly, we define write and call nodes; the type of the exit node is tl. The remaining nodes are categorized as either unconditional assignment (ua) or branch (br) nodes depending on whether there is more than one guarded assignment edge outgoing from n. The node types are well defined due to the fact that the graphs are deterministic and read, write, and call edges are implicitly conditioned on $true$.

Definition 3. *Given a program $\mathcal{S}$, we define a set of* cut points*, denoted $\mathcal{P}^S$, to be a subset of program nodes such that $\mathcal{P}^S = \{\ n^S\ :\ n^S \in \mathcal{N}^S\ \wedge\ \tau(n^S) \neq ua\ \}$.*

Essentially, all the nodes except for the unconditional assignments are added to the cut point set. Ideally, we would like to be able to control the granularity of the cut point set. For example, we may choose to either place one cut point per each loop or one cut point per each branch instruction. This can be achieved through the choice of the transition graph nodes. If we choose $\mathcal{N}^S$ to be the minimal set of procedure locations satisfying the requirements presented at the end of Section 2.2.1, each branch node will cut one loop. Alternatively, one may choose $\mathcal{N}^S$ so that there is a cut at each program conditional. Finer granularity improves efficiency; but it is not always applicable: the input programs have to be consonant modulo the chosen cut point set. In addition, the optimizations that do not preserve the procedural structure of the programs such as inlining or tail recursion elimination (see Section 2.3) require an amendment to the definition of the cut point set. Specifically, the set of target cut points would have to be extended. For example, in case of inlining, it has to include the nodes right before and right after the inlined call sites.

Every computation σ^S defines a corresponding sequence of cut points, which can be obtained from σ^S by first selecting the nodes of each subsequent state (recall that a state is a pair consisting of a node and the data interpretation) and then removing nodes that are not in $\mathcal{P}^S$ from that sequence.

Definition 4. *We say that transition graphs f^S and f^T, which belong to programs $\mathcal{S}$ and $\mathcal{T}$ respectively, are* consonant *if the following two requirements are met.*

- First, there exists a partial map $\kappa \ : \ \mathcal{P}^T \mapsto \mathcal{P}^S$ such that $\forall\, \sigma^S, \sigma^T : \ \sigma^S \in Cmp(f^S), \ \sigma^T \in Cmp(f^T)$ the following holds: if σ^S and σ^T are defined by the same input sequence, and $n_0^S, n_1^S, ...$ and $n_0^T, n_1^T, ...$ are the cut point sequences defined by σ^S and σ^T, then $(\kappa(n_i^T) = n_i^S) \wedge (\tau(n_i^S) = \tau(n_i^T))$, i.e. the i^{th} target cut point is mapped to the i^{th} source cut point and both have the same type. Such map is called a *control abstraction*.

- Let *data abstraction* be a set of assertions that relate the essential variables of the source and target systems at the corresponding control states:

  $$\{ \ \alpha_{ij}(\vec{y}^S, \vec{y}^T) \ : \ i \in \mathcal{P}^S \ \wedge \ j \in \mathcal{P}^T \ \wedge \ i = \kappa(j) \ \}$$

  The definition of *inductive* data abstraction is similar to the one of inductive assertion network, presented in Section 2.2.2 with the only difference that it is defined on computations of the two systems. As the second condition of consonance, we require the automatic construction of an inductive data abstraction to be feasible.

Definition 5. *Two programs are consonant if their main procedures are consonant.*

Note that this definition talks about the computations of the program rather then its control flow graph (or its structure).

Surprisingly many compiler optimizations preserve consonance of programs. For example, code motion, constant folding, reassociation, common subexpression elimination, dead code elimination, instruction scheduling, branch optimizations all fall into this category. On the other hand, loop reordering transformations such as tiling and interchange are not covered by the method presented below.

2.2.5 Notation

We use $\mathcal{S}$ and $\mathcal{T}$ to denote the source and the target programs, respectively. We use lowercase Latin letters to denote the names of the target procedures and variables. We use the uppercase letters to denote the corresponding names in the source program. We may also use the superscript notation to differentiate between the source and target variables, nodes, or transition graphs. The subscript notation is used to associate variables, nodes, or edges with a particular procedure. To simplify the presentation, we are going to use the notation E_g^T to denote $E^T(\vec{y}_g^T)$ - a vector of target program expressions over all the variables of the target procedure g.

2.3 Interprocedural Translation Validation

The Interprocedural Translation Validation framework (ITV) is an extension of TVOC – the deductive method for automatic translation validation presented in [21, 67, 68, 39]. The backbone of TVOC is the Validate rule, which is a refinement rule based on the computational induction approach [24]. Given two input programs: the source program $\mathcal{S}$ and the target program $\mathcal{T}$, we first have to construct a data abstraction and a control abstraction that show the correspondence between the variables and the nodes of $\mathcal{S}$ and $\mathcal{T}$. Further, a set of verification conditions is generated. These conditions use induction to show that the data abstraction is valid and that the data abstraction implies the equivalence of $\mathcal{S}$ and $\mathcal{T}$.

The main restriction of the rule is that it assumes that the two input programs are consonant (see Section 2.2.4). Most of the classical compiler optimizations such as constant folding, induction variable optimizations, branch optimizations, common subexpression elimination, and others, preserve consonance of the programs. Rules for loop reordering transformations [39, 68] can be additionally applied to verify transformations such as loop interchange, fusion, distribution and tiling.

When dealing with a consonant pair of programs, ITV has all the capabilities of TVOC. Additionally, it addresses the following two limitations. First, TVOC only deals with deterministic systems, where the initial condition uniquely determines the rest of the computation. Second, it does not support interprocedural optimizations. In contrast to TVOC, which used transition systems as the formal model, ITV relies on the transition graphs (see Section 2.2.1) that capture not only conditions and assignments but also procedure calls and read/write operations. Note that the set of represented programs is not limited to the deterministic programs, as before, but includes reactive systems driven by intermediate inputs. In addition, the notion of correct translation, defined in Section 2.2.3, allows us to observe the system in the intermediate states. We show how to generate the auxiliary invariants used for verification of context sensitive copy propagation and present the interprocedural translation validation algorithm that proves correct translation of $\mathcal{S}$ to $\mathcal{T}$ in presence of interprocedural optimizations like global constant propagation, inlining, tail-recursion elimination, interprocedural dead code elimination, dead argument elimination, and cloning. Our algorithm is strong enough to handle most, if not all, of the interprocedural optimizations described in literature [47, 10] and performed by compilers such as GCC[6], ORC[2], and LLVM[7].

We start by presenting the outline of the translation validation algorithm. In Section 2.3.2, we show how to generate the auxiliary program invariants required for proving the context sensitive constant copy propagation. Section 2.3.3 lists the translation verification conditions. It also describes how to handle inlining and tail recursion elimination. Finally, Section 2.3.4 is dedicated to an elaborate example.

2.3.1 Interprocedural Translation Validation Algorithm

The Interprocedural Translation Validation algorithm is an extension of the rule Validate [68] to reactive procedural programs. Given two procedural programs $\mathcal{S}$ and $\mathcal{T}$, the algorithm generates a proof that the target program $\mathcal{T}$ is a correct translation of the source program $\mathcal{S}$.

Let $\mathcal{P}^T$ and $\mathcal{P}^S$ denote the sets of cut-points of $\mathcal{T}$ and $\mathcal{S}$ respectively. Assume for now that these cut point sets are have been extended to accommodate the interprocedural optimizations such as inlining and tail recursion elimination. As will be described later, the cut point sets are built with assistance from the compiler. We follow the five steps below to either generate a proof that $\mathcal{T}$ is a correct translation of $\mathcal{S}$ or report a miscompilation.

Step 1: Establish *control abstraction* $\kappa : \mathcal{P}^T \mapsto \mathcal{P}^S$, mapping the target nodes to the source nodes, such that r is the initial location (root of the main procedure) of $\mathcal{T}$ if and only if $\kappa(r)$ is the initial location of $\mathcal{S}$. The mapping κ is total but may be neither surjective nor injective. For example, we allow a non-surjective mapping to handle a situation when a loop is eliminated as part of dead code elimination. Optimizations such as inlining result in a non-injective control abstraction. Note that the control abstraction not only specifies the mapping between the program locations but also imposes a correspondence between target and source procedures. For example, consider a target procedure g with the root node r and the tail node t; g corresponds to source procedure G with the root node $\kappa(r)$ and the tail node $\kappa(t)$.

Step 2: Construct sets of target and source auxiliary assertions that form inductive networks $\Phi^{\mathcal{T}} = \{\varphi_0^T, \ldots, \varphi_{|\mathcal{N}^{\mathcal{T}}|}^T\}$ and $\Phi^{\mathcal{S}} = \{\varphi_0^S, \ldots, \varphi_{|\mathcal{N}^{\mathcal{S}}|}^S\}$ for programs $\mathcal{T}$ and $\mathcal{S}$, respectively. Form verification conditions showing that the networks are invariant, following rules from Section 2.2.2. Add the generated conditions to the set of verification conditions $\mathcal{VC}$.

Step 3: Let $V^{\mathcal{S}}$ and $V^{\mathcal{T}}$ denote the sets of variables that belong to programs $\mathcal{S}$ and $\mathcal{T}$, respectively. Form *data abstraction* $\{\alpha_0, \ldots, \alpha_{|\mathcal{P}^T|}\}$. Each $\alpha_l(V^{\mathcal{S}}; V^{\mathcal{T}})$ is defined as a conjunction of equalities of the form $E(V^{\mathcal{S}}) = E(V^{\mathcal{T}})$ associated with the target node $l \in \mathcal{P}^T$. The data abstraction must be valid at the initial location of $\mathcal{T}$, i.e. $\alpha_r = \mathit{true}$. Intuitively, the data abstraction maps the values of target variables at location l to the values of source variables at location $\kappa(l)$.

Step 4: Form *Translation Verification Conditions*, presented in Section 2.3.3, for every edge of the target program and add them to the set of verification conditions $\mathcal{VC}$. If there exists an edge of the target program that does not contribute a verification condition, generate Error.

Step 5: Establish validity of the conditions in $\mathcal{VC}$; generate Error otherwise. The Error signifies that either an error in translation is detected or we ran into a transformation that is not currently supported.

We rely on the compiler to provide the information necessary to build the data and control abstraction. Methods for abstraction generation are presented in [67]. They rely on the compiler annotations that are usually required for debugging compiled code and are provided by many mature compilers. Construction of the data abstraction is based on refining the mapping between source and target program variables. To construct the control abstraction κ, we first generate the set of source cut-points $\mathcal{P}^S$ such that they satisfy the minimal requirements stated in Section 2.2.1. Then, we rely on the compiler annotations to assist in computation of the control abstraction κ and $\mathcal{P}^T$ by providing the mapping from the source program locations to the target program locations. Finally, we check the $\mathcal{P}^T$ for completeness with respect to the requirements of Section 2.2.1.

2.3.2 Invariants for Interprocedural Constant Propagation

Recall that ***S*tep 2** of the algorithm above calls for construction of the auxiliary assertion set, which is used to strengthen the translation verification conditions from ***Step 4***. This set should contain all the assertions that are used by the TVOC framework. The TVOC's method for invariant generation, presented in [22], is based on the set of reachable definitions (variable definitions that must hold at a particular location) and is applied in the intraprocedural setting. However, the source network has to be extended so that it incorporates the information essential for proving interprocedural optimizations. In this section, we present a method for generating an inductive assertion network that is strong enough to prove interprocedural context sensitive constant copy propagation. Linear constant propagation can be handled in a similar fashion. We are going to use [57] as our interprocedural dataflow analysis algorithm. The algorithm is *precise* and has an efficient representation for the internal data that we can use to our advantage.

As a first try, it appears that any precise solution to the interprocedural constant-propagation problem should suffice. For example, φ_l^S should be extended with conjunct $x = 17$ if x always evaluates to constant 17 at at location l. However, the resulting network $\Phi^{\mathcal{S}}$ may not be inductive. Fortunately, the fixpoint based dataflow analysis algorithm not only provides a solution, but also finds a fixpoint for the corresponding set of dataflow equations. We are going to use the information about the fixpoint itself to strengthen our network so it would be inductive.

Let V be the finite set of program variables. Let $L = Z_{\perp}^{\top}$ be the integer constant propagation lattice. We denote the meet operator by $\sqcap$. The set $Env(V, L)$ of **environments** is the set of functions from V to L. A mapping $T : Env(V, L) \mapsto Env(V, L)$ is called *an environment transformer*. A transformer T is **distributive** iff for every variable $v \in V$, $(T(\sqcap_i env_i))(v) = \sqcap_i (T(env_i))(v)$. The algorithm in [57] essentially computes a transformer $T_{(r_f,l)}$ between the root of each procedure f

and every location of the procedure. Note that the transformer $T_{(r_f,t_f)}$ between the root and the tail of f is essentially a procedure summary that is represented in our framework by the invariant q_f.

Since T needs to operate on functions with infinite domains, the following succinct representation for distributive transformers is used in [57]. Every distributive transformer T can be represented using a set of functions $\Omega^T = \{\rho_{v,v'} \mid v, v' \in V \cup \{\Lambda\}\}$, each of type $L \mapsto L$. Function $\rho_{v,v'}$ captures the effect that the value of variable v in the argument environment has on the value of v' in the result environment; if v' does not depend on v, then $\rho_{v,v'} = \lambda l.\top$. Function $\rho_{\Lambda,v'}$ is used to represent the effect on the variable v that is independent of the argument environment. For any symbol v', the value $T(env)(v')$ can be determined by taking the meet of the values of $|V|+1$ individual function applications: $T(env)(v') = \rho_{\Lambda,v'} \sqcap (\sqcap_{v\in V} \rho_{v,v'}((env)(v)))$. Since we are only concerned with constant copy propagation, all the functions in Ω^T will be either identities or constants.

Consider the example in Fig. 2.5. Below is the list of environment transformers computed by [57] for procedure foo. We omit all the functions that evaluate to top $\rho_{(v,v')} = \lambda l.\top$.

$$\begin{aligned}
\Omega_{(2,2)} = &\quad \{\ \rho_{x,x} = \lambda l.l,\ \rho_{c,c} = \lambda l.l,\ \rho_{y,y} = \lambda l.l,\ \rho_{z,z} = \lambda l.l\ \} \\
\Omega_{(2,3)} = &\quad \{\ \rho_{c,c} = \lambda l.l,\ \rho_{y,y} = \lambda l.l,\ \rho_{y,z} = \lambda l.l\ \} \\
\Omega_{(2,4)} = &\quad \{\ \rho_{c,c} = \lambda l.l,\ \rho_{c,z} = \lambda l.l,\ \rho_{y,z} = \lambda l.l\ \}
\end{aligned}$$

Given all the dataflow facts (constants) and the transformer represented by $\Omega_{(i,j)}$, we follow the following rules to compute an invariant φ_l at location l of f:

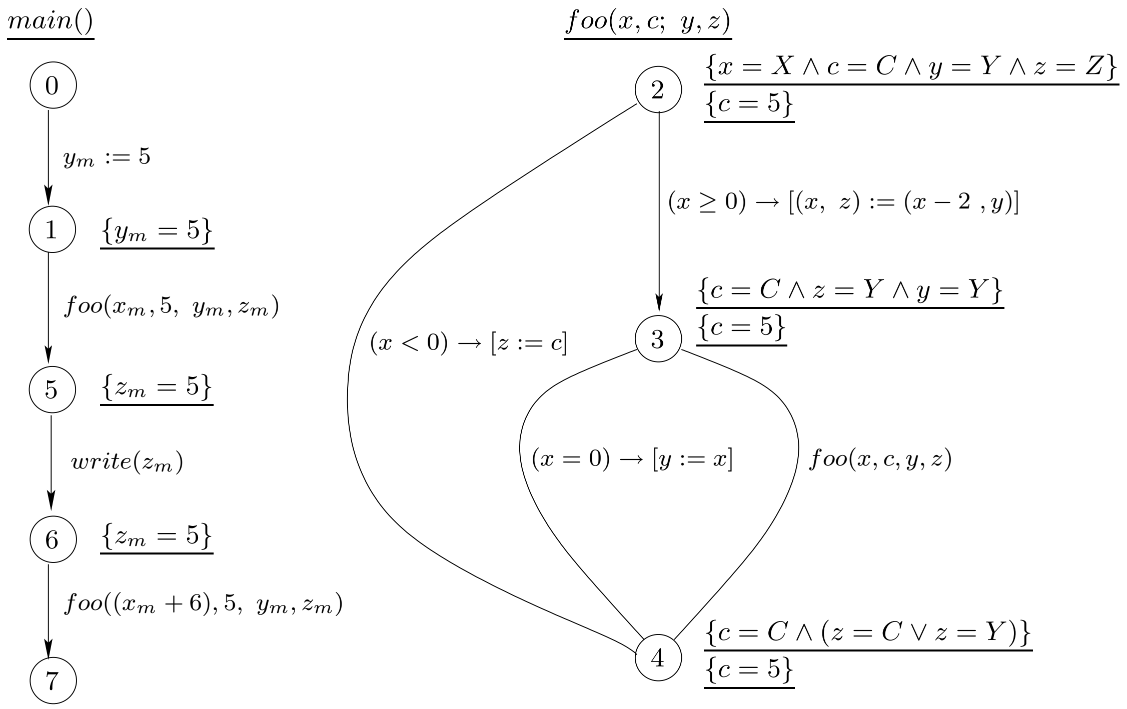
- We ignore all functions of the form $\rho_{(v,v')} = \lambda l.\top$.

- For each variable v' that is not set to $\perp$ by $\rho_{(\Lambda,v')} \in \Omega_{(r_f,l)}$ we add the following conjunct to φ_l:

  $$\bigvee_{\rho_{v,v'} \in \Omega_{(r_f,l)}} v' = \rho_{v,v'}\ (V), \text{ where } V \text{ is the value of } v \text{ at the procedure entry.}$$

  We use disjunction to model the effect of the meet operator. In our example, we use fictitious variables X, C, Y, Z to store the the initial values of x, c, y, z.

- We also add the conjunct $x = \mathit{const}$ if x was determined to evaluate to constant *const* at location l. We need this addition since $T_{(r_f,l)}$ does not propagate the information from the callers.



$\underline{main()}$
0
$y_m := 5$
1
$\{y_m = 5\}$
$foo(x_m, 5,\ y_m, z_m)$
5
$\{z_m = 5\}$
$write(z_m)$
6
$\{z_m = 5\}$
$foo((x_m + 6), 5,\ y_m, z_m)$
7
$\underline{foo(x, c;\ y, z)}$
2
$\{x = X \wedge c = C \wedge y = Y \wedge z = Z\}$
$\{c = 5\}$
$(x \geq 0) \rightarrow [(x,\ z) := (x - 2\ , y)]$
3
$\{c = C \wedge z = Y \wedge y = Y\}$
$\{c = 5\}$
$(x < 0) \rightarrow [z := c]$
$(x = 0) \rightarrow [y := x]$
$foo(x, c, y, z)$
4
$\{c = C \wedge (z = C \vee z = Y)\}$
$\{c = 5\}$

Figure 2.5: Inductive network for interprocedural constant copy propagation

The resulting invariants, denoted in Fig. 2.5 by curly brackets, form an inductive network. For example, let's show that the return verification condition for call edge $(1,5)$ of our example holds.

$$\begin{array}{lllllll} \mathcal{VC}_{ret}: & \varphi_1 & \wedge & \varphi_4[(C,Y) \mapsto (5, y_m)] & \rightarrow & \varphi_5[z_m \mapsto z] & \Leftrightarrow \\ & y_m = 5 & \wedge & c = 5 \wedge (z = 5 \vee z = y_m) \wedge c = 5 & \rightarrow & z = 5 & \end{array}$$

2.3.3 Translation Verification Conditions

Similarly to the verification conditions used to prove the assertion network inductive (see Section 2.2.2), Translation Verification Conditions prove that the data abstraction is inductive on the computations of the target program. They also ensure that source and target observations match given the consistent input.

We first give a recipe for generating translation verification conditions when the structure of the transformed program is preserved: for every edge of the target program $e^{\mathcal{T}}$ connecting nodes i and j, there exist the corresponding source edge $e^{\mathcal{S}}$ between nodes $\kappa(i)$ and $\kappa(j)$:

- *Guarded Assignment*: If the target edge $e^{\mathcal{T}}$ is a guarded assignment edge of $\mathcal{T}$; and $\kappa(i), \kappa(j)$ are also connected by one or more assignment edges in $\mathcal{S}$, we generate the following conditions.

$$\boxed{\begin{array}{l} \alpha_i \wedge \varphi_i^S \wedge \varphi_i^T \wedge \rho_{e^{\mathcal{T}}} \rightarrow (\bigvee_{e^{\mathcal{S}} \in Edges(\kappa(i),\kappa(j))} c_{e^{\mathcal{S}}}) \\ \alpha_i \wedge \varphi_i^S \wedge \varphi_i^T \wedge \rho_{e^{\mathcal{T}}} \wedge (\bigvee_{e^{\mathcal{S}} \in Edges(\kappa(i),\kappa(j))} \rho_{e^{\mathcal{S}}}) \rightarrow \alpha\prime_j \end{array}}$$

In the formulas above, for an edge $e \in \{e^{\mathcal{S}}, e^{\mathcal{T}}\}$ labeled by $c \rightarrow [\vec{u} := E(\vec{y})]$, c_e stands for the condition c and ρ_e for the expression $c \wedge (\vec{u}' = E(\vec{y})) \wedge \vec{v}' = \vec{v}$, where $\vec{v}$ are all variables of α_j with the exception of those in $\vec{u}$. The first implication checks that whenever the target transition is enabled, at least one of the corresponding source transitions is also enabled. The second verification condition checks that the data abstraction is preserved by the matching target and source transitions. Invariants φ_i^S and φ_i^T are used to strengthen the left-hand-side of the implication.

- *Read*: If $e^{\mathcal{T}}$ and $e^{\mathcal{S}}$ are both labeled by read instructions $read(\vec{u}^T)$ and $read(\vec{u}^S)$, the following condition is generated.

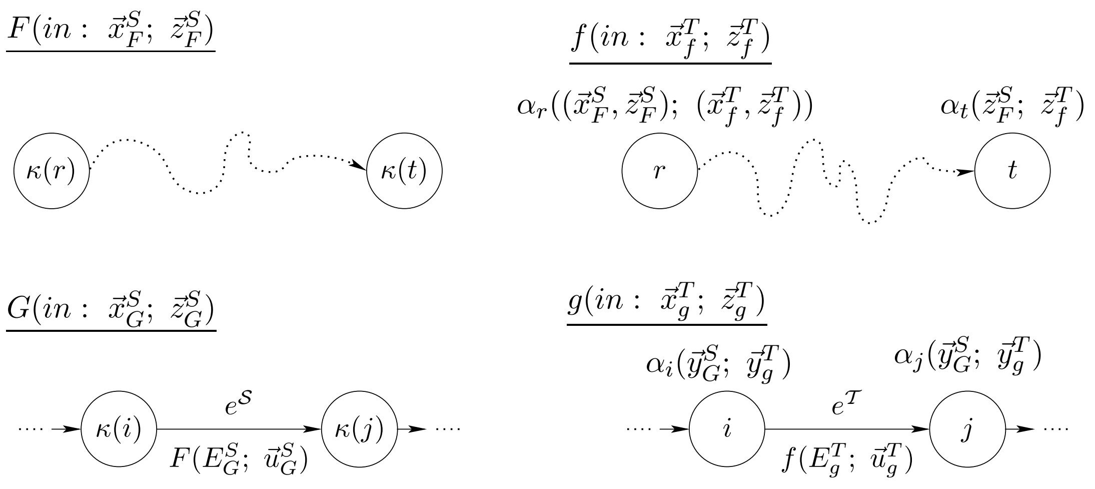
$$\boxed{\alpha_i \wedge \varphi_i^T \wedge \varphi_{\kappa(i)}^S \wedge (\vec{u}^T = \vec{u}^S) \rightarrow \alpha_j}$$

- *Write*: If $e^{\mathcal{T}}$ and $e^{\mathcal{S}}$ are both write edges, labeled by $write(E^T(\vec{y}^T))$ and $write(E^S(\vec{y}^S))$, we add the following implication to the set of verification conditions.

$$\boxed{\alpha_i \wedge \varphi_i^T \wedge \varphi_{\kappa(i)}^S \rightarrow \alpha_j \wedge (E^T(\vec{y}^T) = E^S(\vec{y}^S))}$$

Read and write verification conditions ensure that the data mapping implies matching source and target output given the consistent input.

- *Procedure Call*: If both $e^{\mathcal{T}}$ and $e^{\mathcal{S}}$ are call edges labeled by $f(\ E_g^T;\ \vec{u}_g^T\ )$ and $F(\ E_G^S;\ \vec{u}_G^S\ )$, respectively, where the target procedure f is mapped to


$F(in:\ \vec{x}^S_F;\ \vec{z}^S_F)$
$\kappa(r)$
$\kappa(t)$
$f(in:\ \vec{x}^T_f;\ \vec{z}^T_f)$
$\alpha_r((\vec{x}^S_F, \vec{z}^S_F);\ (\vec{x}^T_f, \vec{z}^T_f))$
$\alpha_t(\vec{z}^S_F;\ \vec{z}^T_f)$
r
t
$G(in:\ \vec{x}^S_G;\ \vec{z}^S_G)$
$\kappa(i)$
$e^{\mathcal{S}}$
$F(E^S_G;\ \vec{u}^S_G)$
$\kappa(j)$
$g(in:\ \vec{x}^T_g;\ \vec{z}^T_g)$
$\alpha_i(\vec{y}^S_G;\ \vec{y}^T_g)$
$\alpha_j(\vec{y}^S_G;\ \vec{y}^T_g)$
i
$e^{\mathcal{T}}$
j
$f(E^T_g;\ \vec{u}^T_g)$

$\mathcal{VC}_{call}$:

$$\alpha_i(\vec{y}^S_G;\ \vec{y}^T_g)\ \wedge\ \varphi^S_{\kappa(i)}(\vec{y}^S_G)\qquad \wedge\ \varphi^T_i(\vec{y}^T_g)\qquad \rightarrow\ \alpha_r((E^S_G, \vec{u}^S_G);\ (E^T_g, \vec{u}^T_g))$$

$\mathcal{VC}_{ret}$:

$$\left[\begin{array}{lllll} \alpha_i(\vec{y}^S_G;\ \vec{y}^T_g) & \wedge & \varphi^S_{\kappa(i)}(\vec{y}^S_G) & \wedge & \varphi^T_i(\vec{y}^T_g)\ \wedge \\ \alpha_t(\vec{z}^S_F;\ \vec{z}^T_f) & \wedge & q^S_F(E^S_G, \vec{u}^S_G;\ \vec{z}^S_F) & \wedge & q^T_f(E^T_g, \vec{u}^T_g;\ \vec{z}^T_f) \end{array}\right] \rightarrow \alpha_j(\vec{y}^S_G[\vec{u}^S_G \mapsto \vec{z}^S_F];\ \vec{y}^T_g[\vec{u}^T_g \mapsto \vec{z}^T_f])$$

In the above formulas, $\alpha_j(\vec{y}^S_G[\vec{u}^S_G \mapsto \vec{z}^S_F]; \vec{y}^T_g[\vec{u}^T_g \mapsto \vec{z}^T_f])$ is obtained from $\alpha_j(\vec{y}^S_G; \vec{y}^T_g)$ after replacing the variables in $\vec{u}^S_G$ by the corresponding variables in $\vec{z}^S_F$ and variables in $\vec{u}^T_g$ by the corresponding variables in $\vec{z}^T_f$.

Figure 2.6: Call verification conditions

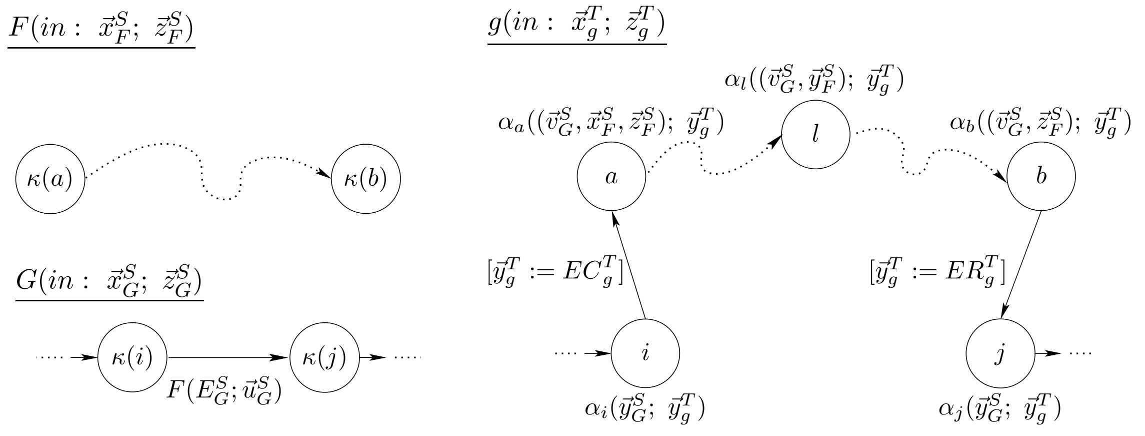
the source procedure F, we generate Call Verification Conditions presented in Fig. 2.6. In this case, procedure G calls procedure F in the source program and procedure g calls procedure f in the target program. Note that the data abstraction associated with the procedure entry $\alpha_r((\vec{x}^S_F, \vec{z}^S_F);\ (\vec{x}^T_f, \vec{z}^T_f))$ depends only on the input variables of the source procedure F and the target procedure f. Similarly, the data abstraction associated with the procedure tail $\alpha_t(\vec{z}^S_F;\ \vec{z}^T_f)$ only depends on the return variables of the two corresponding procedures.

The Call Verification Conditions check that the data abstraction is preserved after stepping through the procedure calls. Similarly to the call conditions of Section 2.2.2, the $\mathcal{VC}_{call}$ condition checks that the data mapping holds at the entry to the procedure; and the $\mathcal{VC}_{ret}$ condition guarantees that it holds after the procedure return. Consider the $\mathcal{VC}_{ret}$ condition. Here, we use the information about the data relation before the procedure call (stored in α_i) and the relation that holds when the programs are about to exit from the calls (stored in α_t) to imply α_j – the data abstractions after the call. We have to make sure that we use the most recent data about the variables passed by reference. This is why we perform the substitution of the variables in α_t. Finally, the right-hand-sides of the implications are strengthened by the auxiliary invariants of the source and target systems; recall that q_F^S and q_f^T are the output predicates of F and f, respectively.

If procedure f is not mapped to procedure F, the algorithm raises the `Error`.

Inlining and Tail-Recursion Elimination(TRE) introduce situations in which the source code contains a call edge that corresponds to a subgraph in the target. In this case, we prove the translation by "stepping into" the procedure call on the source. Let $e^{\mathcal{T}} = (i, a)$ be an unconditional assignment edge of the target such that there exists a source call edge $(\kappa(i), \kappa(j))$, labeled by $F(E_G^S; \vec{u}_G^S)$; $\kappa(a)$ is the entry node of F; and there exists the corresponding node b in the target such that $\kappa(b)$ is the exit node of the procedure F. If (b, j) is an unconditional assignment of $\mathcal{T}$, proceed with inlining verification conditions; otherwise, consider TRE.

Inlining



$F(in:\ \vec{x}^S_F;\ \vec{z}^S_F)$
$\kappa(a)$
$\kappa(b)$
$G(in:\ \vec{x}^S_G;\ \vec{z}^S_G)$
$\kappa(i)$
$\kappa(j)$
$F(E^S_G; \vec{u}^S_G)$
$g(in:\ \vec{x}^T_g;\ \vec{z}^T_g)$
$\alpha_l((\vec{v}^S_G, \vec{y}^S_F);\ \vec{y}^T_g)$
$\alpha_a((\vec{v}^S_G, \vec{x}^S_F, \vec{z}^S_F);\ \vec{y}^T_g)$
l
$\alpha_b((\vec{v}^S_G, \vec{z}^S_F);\ \vec{y}^T_g)$
a
b
$[\vec{y}^T_g := EC^T_g]$
$[\vec{y}^T_g := ER^T_g]$
i
j
$\alpha_i(\vec{y}^S_G;\ \vec{y}^T_g)$
$\alpha_j(\vec{y}^S_G;\ \vec{y}^T_g)$

$\mathcal{VC}_{call}$:

$$\alpha_i(\vec{y}^S_G;\ \vec{y}^T_g) \quad \wedge \quad \varphi^T_i(\vec{y}^T_g) \quad \wedge \quad \varphi^S_{\kappa(i)}(\vec{y}^S_G) \quad \rightarrow \quad \alpha_a((\vec{v}^S_G, E^S_G, \vec{u}^S_G); EC^T_g)$$

$\mathcal{VC}_{ret}$:

$$\alpha_b((\vec{v}^S_G, \vec{z}^S_F);\ \vec{y}^T_g) \quad \wedge \quad \varphi^T_b(\vec{y}^T_g) \quad \wedge \quad \varphi^S_{\kappa(i)}(\vec{y}^S_G) \quad \wedge \quad q^S_F(E^S_G, \vec{u}^S_G; \vec{z}^S_F) \quad \rightarrow \quad \alpha_j(\vec{y}^S_G[\vec{u}^S_G \mapsto \vec{z}^S_F]; ER^T_g)$$

where $\vec{v}^S_G \ = \ \vec{y}^S_G \setminus \vec{u}^S_G$ – are the variables of G excluding the ones passed by reference to F.

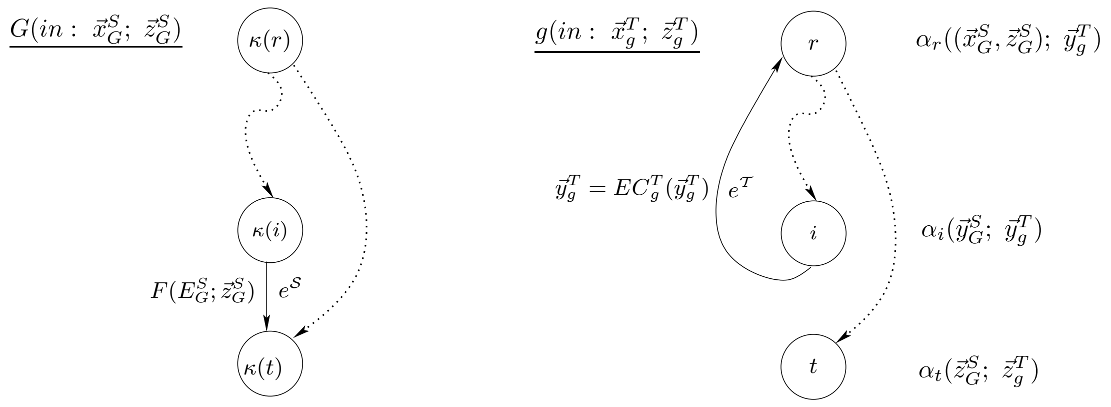
Figure 2.7: Inlining verification conditions

Consider the case depicted in Fig. 2.7 where a source call edge $e^{\mathcal{S}} = (\kappa(i), \kappa(j))$, labeled by $F(E^S_G,\ \vec{u}^S_G)$, has been inlined. Suppose that the target locations i and j belong to some procedure g. To simplify this presentation, we assume that there is no nested inlining, so $e^{\mathcal{S}}$ belongs to G such that g is mapped to G. The target procedure should contain unconditional assignment transitions (i, a) and (b, j) that correspond to the call to and return from procedure F on the source. Assume, (i, a) is labeled by $[\vec{y}^T_g := EC^T_g]$ and (b, j) is labeled by $[\vec{y}^T_g := ER^T_g]$.

Define a set of target locations $L \subset \mathcal{P}^T$ such that it includes all locations on every path from i to j. Note that all the locations in this set will be mapped to the nodes of the source procedure F. It is required that α_l, $l \in L$ does not depend on $\vec{u}_G^S$, the variables whose references are passed to F. However, we do allow the dependance on the corresponding formal parameters. This restriction comes from the fact that $\vec{u}_G^S$ may change during the execution of F. Inlining Verification Conditions, presented in Fig. 2.7, are generated for each pair of target locations (i, j) that correspond to the inlined call edge $(\kappa(i), \kappa(j))$. The $\mathcal{VC}_{call}$ condition checks the data abstraction associated with locations a and $\kappa(a)$, which are reached after the assignment on the target and the call on the source; the $\mathcal{VC}_{ret}$ condition checks that the data abstraction holds at locations j and $\kappa(j)$ – after the corresponding assignment and the return. This ensures that both of the target edges, (i, a) and (b, j), contribute a condition to the set $\mathcal{VC}$.

Tail Recursion Elimination

A call edge (i, t) of a procedure $f(\vec{x};\ \&\vec{z})$ is a **TRE candidate** if it is a recursive call labeled by $f(E(y); \vec{z})$ and t is the tail node of procedure f. Note that the formal output parameters $\vec{z}$ are passed as the actual parameters in the tail call. Let $e^{\mathcal{T}} = (i, r)$ be an unconditional assignment edge of the target procedure g such that there exists a TRE candidate source edge $(\kappa(i), \kappa(t))$ labeled by $G(E_G^S(\vec{y}_G^S), \vec{z}_G^S)$, where the target procedure g is mapped to the source procedure G. Under these conditions, we guess that TRE optimization occurred and generate TRE Verification Condition, shown in Fig. 2.8. The condition checks that the data abstraction holds at the entry to the procedure: after a call on the source and the assignment on the


$G(in: \vec{x}^S_G;\ \vec{z}^S_G)$
$\kappa(r)$
$\kappa(i)$
$F(E^S_G; \vec{z}^S_G)$
$e^{\mathcal{S}}$
$\kappa(t)$
$g(in: \vec{x}^T_g;\ \vec{z}^T_g)$
r
$\alpha_r((\vec{x}^S_G, \vec{z}^S_G);\ \vec{y}^T_g)$
$\vec{y}^T_g = EC^T_g(\vec{y}^T_g)$
$e^{\mathcal{T}}$
i
$\alpha_i(\vec{y}^S_G;\ \vec{y}^T_g)$
t
$\alpha_t(\vec{z}^S_G;\ \vec{z}^T_g)$

$$\boxed{\mathcal{VC}_{call:}\ \alpha_i(\vec{y}^S_G;\ \vec{y}^T_g) \ \wedge\ \varphi^T_i(\vec{y}^T_g) \ \wedge\ \varphi^S_{\kappa(i)}(\vec{y}^S_G) \ \rightarrow\ \alpha_r((E^S_G, \vec{z}^S_G);\ EC^T_g(\vec{y}^T_g))}$$

Figure 2.8: Tail recursion elimination verification conditions

target are performed. There is no target edge that corresponds to the return from the recursive call on the source and, consequently, the exit verification condition is not generated. Next, we explain why the requirements of the correct translation, as defined in Section 2.2.3, are still satisfied. Consider a source computation $\sigma_{\mathcal{S}}$ that contains m recursive calls to G and the corresponding target computation $\sigma_{\mathcal{T}}$:

$$\begin{array}{ll}
\sigma_{\mathcal{S}} = & \dots \\
& \langle \kappa(r),\ D^0_r \rangle \longrightarrow \dots \longrightarrow \langle \kappa(i),\ D^0_i \rangle \xrightarrow{e^{\mathcal{S}}} \\
& \dots \\
& \langle \kappa(r),\ D^{m-1}_r \rangle \longrightarrow \dots \longrightarrow \langle \kappa(i),\ D^{m-1}_i \rangle \xrightarrow{e^{\mathcal{S}}} \\
& \langle \kappa(r),\ D^m_r \rangle \longrightarrow \dots \longrightarrow \langle \kappa(t),\ D^m_t \rangle \\
& \xrightarrow{ret} \langle \kappa(t),\ D^{m-1}_t \rangle \ \dots\ \xrightarrow{ret} \langle \kappa(t),\ D^1_t \rangle \xrightarrow{ret} \langle \kappa(t),\ D^0_t \rangle \ \dots
\end{array}$$

$$\begin{array}{lllll}
\sigma_{\mathcal{T}} = & \ldots & & & \\
 & \langle r,\ d_r^0 \rangle & \longrightarrow \ldots \longrightarrow & \langle i,\ d_i^0 \rangle & \xrightarrow{e^{\mathcal{T}}} \\
 & \ldots & & & \\
 & \langle r,\ d_r^{m-1} \rangle & \longrightarrow \ldots \longrightarrow & \langle i,\ d_i^{m-1} \rangle & \xrightarrow{e^{\mathcal{T}}} \\
 & \langle r,\ d_r^{m} \rangle & \longrightarrow \ldots \longrightarrow & \langle t,\ d_t^{m} \rangle & \\
 & \ldots & & &
\end{array}$$

D_l^k and d_l^k denote source and target data interpretation, where k stands for the recursion level in the source program and the iteration level in the target program.

First, we want to show that $\alpha_t(\vec{z}_G^S;\ \vec{z}_g^T)$ holds. Validity of the verification conditions generated for all target edges that end in t prove that $\alpha_t(D_t^m;\ d_t^m)$ holds: α_t holds before we take the very first return transition in the source. Note that the only source variables effecting $\alpha_t(\vec{z}_G^S;\ \vec{z}_g^T)$ are the formal parameters passed by reference that match the actual parameters used for the tail call. Therefore, popping the stack does not change $\vec{z}_G^S$, and α_t is preserved by the return transitions.

Second, we show that for every target observation, there exists a stuttering equivalent source observation. Consider the observations of the source and target programs $o_{\mathcal{S}}$ and $o_{\mathcal{T}}$ that can be obtained by applying the observation function $\mathcal{O}$ to the computations $\sigma_{\mathcal{S}}$ and $\sigma_{\mathcal{T}}$:

$$
\begin{array}{llllll}
o_{\mathcal{S}} = & \ldots & & & & \\
& \mathcal{O}(\langle \kappa(r),\ D_r^0 \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle \kappa(i),\ D_i^0 \rangle) & \xrightarrow{\top} \\
& \ldots & & & & \\
& \mathcal{O}(\langle \kappa(r),\ D_r^{m-1} \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle \kappa(i),\ D_i^{m-1} \rangle) & \xrightarrow{\top} \\
& \mathcal{O}(\langle \kappa(r),\ D_r^{m} \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle \kappa(t),\ D_t^{m} \rangle) & \\
& \multicolumn{5}{l}{\underbrace{\xrightarrow{\top}\ \top \ldots \xrightarrow{\top} \top\ \xrightarrow{\top} \top}_{m \text{ return transitions}} \ldots}
\end{array}
$$

$$
\begin{array}{llllll}
o_{\mathcal{T}} = & \ldots & & & & \\
& \mathcal{O}(\langle r,\ d_r^0 \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle i,\ d_i^0 \rangle) & \xrightarrow{\top} \\
& \ldots & & & & \\
& \mathcal{O}(\langle r,\ d_r^{m-1} \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle i,\ d_i^{m-1} \rangle) & \xrightarrow{\top} \\
& \mathcal{O}(\langle r,\ d_r^{m} \rangle) & \longrightarrow & \ldots \longrightarrow & \mathcal{O}(\langle t,\ d_t^{m} \rangle) & \\
& \ldots & & & &
\end{array}
$$

The verification conditions that we generate for each target edge ensure that for every target transition in $\sigma_{\mathcal{T}}$ there exists a corresponding source transition in $\sigma_{\mathcal{S}}$. Furthermore, the I/O transitions (and the associated data) match. Thus, the source observation $o_{\mathcal{S}}$ can be obtained from the target observation $o_{\mathcal{T}}$ by adding exactly m pairs $\xrightarrow{\top} \top$.

2.3.4 ITV Example: Constant Propagation and TRE

In Fig. 2.9, we show the source and the target programs. These programs output $(3 * A)! + 5$ given an input A. The target program, depicted on the bottom, is obtained from the source after TRE is applied to the procedure $fact$, the value of constant c is propagated, and the computation of the expression $a * 3$ is moved due

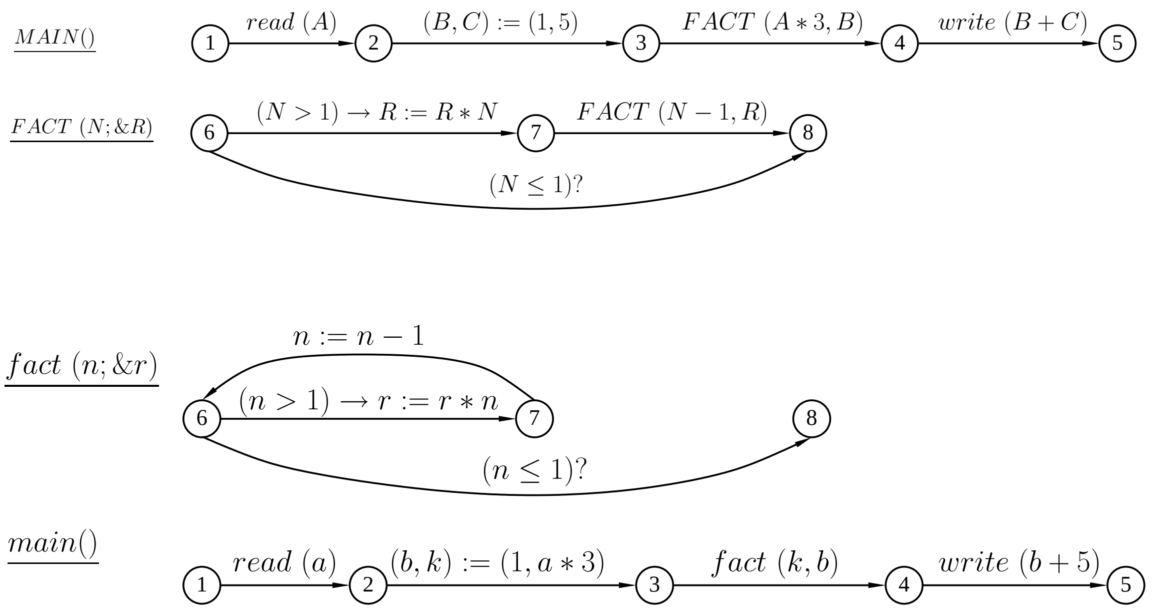


Figure 2.9: ITV example: the program before and after compilation

to instruction scheduling. Note our notation: we use uppercase letters to denote the source variables and procedure names; we use their lowercase counterparts for the target program. Let us apply the Interprocedural Translation Validation (ITV) algorithm from Section 2.3.1 to prove that the target is the correct translation of the source in this example.

Step 1: The control abstraction κ is identity.

Step 2: The inductive assertion network $\Phi^{\mathcal{S}}$ associates assertion $(C = 3)$ with locations $l \in \{3, 4, 5\}$ and assertion *true* with $l \in \{1, 2, 6, 7, 8\}$. All the assertions in $\Phi^{\mathcal{T}}$ are *true*. We omit the set of the verification conditions that prove inductiveness of $\Phi^{\mathcal{S}}$ and $\Phi^{\mathcal{T}}$ since they are straightforward.

Step 3: The following data abstraction is generated:

$$\begin{array}{lll}
\alpha_1 & : & true \\
\alpha_2 & : & (A \; = \; a) \\
\alpha_3 & : & (A * 3 \; = \; k) \quad \wedge (B \; = \; b) \\
\alpha_4 & : & (A * 3 \; = \; k) \quad \wedge (B \; = \; b) \\
\alpha_5 & : & true
\end{array}
\qquad
\begin{array}{lll}
\alpha_6 & : & (R \; = \; r) \quad \wedge (N \; = \; n) \\
\alpha_7 & : & (R \; = \; r) \quad \wedge (N \; = \; n) \\
\alpha_8 & : & (R \; = \; r)
\end{array}$$

Step 4: Below, we list selected translation verification conditions from the set $\mathcal{VC}$. We are going to omit the invariants that evaluate to *true*:

$\underline{Read: \; \mathcal{VC}_{(1,2)}}: \quad \alpha_1 \; \wedge \; (A = a) \; \rightarrow \; \alpha_2 \quad \Leftrightarrow \quad true \; \wedge \; (a = A) \; \rightarrow \; (a = A)$

$\underline{Assign: \; \mathcal{VC}_{(2,3)}}: \quad \alpha_2 \; \wedge \; \rho_{(2,3)^{\mathcal{T}}} \; \rightarrow \; c_{(2,3)^{\mathcal{S}}}; \; \alpha_2 \; \wedge \; \rho_{(2,3)^{\mathcal{T}}} \; \wedge \; \rho_{(2,3)^{\mathcal{S}}} \; \rightarrow \; \alpha'_3 \; \Leftrightarrow$

$(A \; = \; a) \; \wedge \; ((b' \; = \; 1) \; \wedge \; (k' = a * 3) \; \wedge \; (a' \; = \; a)) \; \rightarrow \; true;$

$(A \; = \; a) \; \wedge \; ((b' \; = \; 1) \; \wedge \; (k' = a * 3) \; \wedge \; (a' \; = \; a)) \; \wedge$

$((B' \; = \; 1) \; \wedge \; (C' = 5) \; \wedge \; (A' \; = \; A)) \; \rightarrow \; ((A' * 3 \; = \; k') \; \wedge \; (B' \; = \; b'))$

$\underline{Call: \; \mathcal{VC}_{(3,4)}}: \quad \mathcal{VC}_{call} : \alpha_3 \; \wedge \; \varphi_3^S \; \rightarrow \; \alpha_6[(N, R) \mapsto (A * 3, B); (n, r) \mapsto (k, b)];$

$\mathcal{VC}_{ret} : \alpha_3 \; \wedge \; \varphi_3^S \; \wedge \alpha_8 \rightarrow \; \alpha_4[B \mapsto R; b \mapsto r] \quad \Leftrightarrow$

$((A * 3 \; = \; k) \; \wedge \; (B \; = b)) \; \wedge \; (C = 5) \; \rightarrow \; ((A * 3 = k) \; \wedge \; (B = b));$

$((A * 3 \; = \; k) \; \wedge \; (B \; = b)) \; \wedge \; (C = 5) \; \wedge \; (R = r) \; \rightarrow \; ((A * 3 = k) \; \wedge \; (R = r))$

$\underline{Write: \; \mathcal{VC}_{(4,5)}}: \quad \alpha_4 \; \wedge \; \varphi_4^S \; \rightarrow \; \alpha_5 \; \wedge \; (B + C = b + 5) \quad \Leftrightarrow$

$((A * 3 \; = \; k) \; \wedge \; (B \; = b)) \; \wedge \; (C = 5) \; \rightarrow \; true \; \wedge \; (B + C = b + 5)$

$\underline{TRE: \; \mathcal{VC}_{(7,6)}}: \quad \alpha_7 \; \rightarrow \; \alpha_6[N \mapsto (N - 1); n \mapsto (n - 1)] \quad \Leftrightarrow$

$((R \; = \; r) \; \wedge \; (N \; = n)) \; \rightarrow \; ((R \; = \; r) \; \wedge \; (N - 1 \; = n - 1))$

Step 5: We use an automatic theorem prover, such as [9, 1], to check the generated conditions for validity. Since all the conditions in $\mathcal{VC}$ are valid, we conclude the correctness of the translation.

2.4 Compiler Validation via Analysis of the Cross-Product (CoVaC)

The previous section presented a translation validation algorithm for verification of structure preserving optimizations. However, notice that most of the techniques it used have been borrowed from the existing methodology for proving properties of a single program. The question is: Is it possible to transform the program equivalence problem to checking a property of a single program? If the answer is yes, the many existing and future techniques and tools could be seamlessly incorporated to solve the translation validation problem.

Additionally, notice that the previous algorithm relied on the compiler to provide the information necessary to construct the data abstraction and the control abstraction. However, is it possible to deduce those without any help from the compiler?

This section presents the framework for program equivalence checking using program analysis of the cross-product framework, which addresses the two questions posed above.

2.4.1 Introduction

The Compiler Verification by Program Analysis of the Cross-Product framework (CoVaC) is a novel translation validation approach, in which one constructs a *comparison system* – a cross-product of the source and target programs. The input programs are equivalent if and only if the comparison system satisfies a certain specification. This allows us to leverage the existing methods of proving properties of a single program instead of relying on program analysis and proof rules specialized to translation validation, used by the existing frameworks [67, 51, 59]. CoVaC is not tailored to validation of compiler transformations – it targets program equivalence in general; for example, it can be applied to validation of language-based security properties [13].

The CoVaC framework can be used in various settings, and, while the check for specification conformance is expected to be the same, the construction of the comparison system may diverge. For example, compiler writers may use translation validation for the creation of a self-certifying compiler and, thus, may assume full knowledge of the inner workings of a particular compiler. In this case, the compiler itself may output the comparison system. Section 2.4.3 pursues the other extreme – it describes a method for automatic generation of the comparison system, and thus, a translation validation algorithm, which accommodates no compiler cooperation. To the best of our knowledge, the existing translation validation frameworks which handle a comparable set of optimizations at least to some degree rely on compiler assistance.

The lack of compiler dependency makes it possible to develop a general purpose verification tool that can be used to verify the transformations performed by different compilers. The only module that has to be developed in order to hook up a new compiler to the tool is a translator from the compiler IR to the format accepted by the tool. One advantage here is the reduction of the development time. Furthermore, since our correctness claim relies on faithfulness of the translation validation tool, another advantage is that most of the trusted core is reused. Such tool would be especially useful to compiler users who may have to work with a particular existing compiler. Additionally, this methodology can be of service to compiler developers to facilitate testing of immature compilers. The traditional compiler test procedures are limited to compilation of a program from a test suite; running the compiled program on several fixed input values; and comparing the actual output, produced by the compiler, to the expected one. Application of translation validation to the programs in the test suite gives a much stronger guarantee – each source program is the correct translation of the target only if their outputs match on *all* possible inputs.

In order to make the validator of non-cooperative compilers feasible and effective, we restrict the set of transformations under consideration to intraprocedural optimizations in which each loop in the target program corresponds to a loop in the source program; we refer to such input systems as *consonant*. Many of the classical compiler optimizations such as constant folding, reassociation, induction variable optimizations, common subexpression elimination, code motion, branch optimizations, register allocation, instruction scheduling, and others fall into this category. These optimizations are usually referred to as structure preserving [67]. Finally, we have developed a prototype tool CoVaC that applies the developed framework to verification of optimizing transformations performed by LLVM [7] – an aggressive open-source C and C++ compiler.

In summary, Section 2.4 describes the following contributions. In the next section, we present a novel deductive framework for checking equivalence of infinite state programs. Section 2.4.3 defines the notion of consonance and shows how the method can be effectively applied to consonant programs. The presented algorithm does not rely on any additional input; thus, it can be used to verify compilations while treating the compiler as a black-box. Section 2.4.5 describes the CoVaC tool. It also lists the existing algorithms used by the tool for invariant generation. The CoVaC tool utilizes a special algorithm for proving equivalence of unbounded memory regions, which is presented in Section 2.4.4. Finally, Section 2.4.6 focuses on the experimental results.

2.4.2 Comparison Graphs

In this section, we show that the problem of establishing correct translation is equivalent to construction of a cross-product (comparison) system $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ and checking if $\mathcal{C}$ satisfies a set of correctness conditions. Our framework is general enough for establishing translation correctness of deterministic systems in presence of a wide set of intraprocedural transformations. Later, we present the application of the method to proving translation of consonant systems. However, the general framework can be used to reason about translation correctness in presence of structure modifying optimizations such as loop transformations [68].

Assume we are given two programs, $\mathcal{S}$ and $\mathcal{T}$. For each pair of the corresponding source and target procedures, $f^S = (\vec{y}^S,\ \mathcal{N}^S,\ \mathcal{E}^S)$ and $f^T = (\vec{y}^T,\ \mathcal{N}^T,\ \mathcal{E}^T)$, a graph satisfying the set of rules below is called a *comparison transition graph*, denoted $f = (\vec{y},\ \mathcal{N},\ \mathcal{E}) = f^S \boxtimes f^T$. f represents a simultaneous execution of f^S and f^T. The collection of comparison graphs for all procedures constitutes the comparison program $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$.

***Rule* 1.** *(Structural Requirement)*

1. *The variables of the comparison graph $\vec{y} = (\vec{x}, \vec{z}, \vec{w})$ are defined as follows: $\vec{x} = \vec{x}^S \circ \vec{x}^T$; $\vec{z} = \vec{z}^S \circ \vec{z}^T$; and $\vec{w} = \vec{w}^S \circ \vec{w}^T$, where $\vec{v} \circ \vec{u}$ denotes concatenation of two vectors.*

2. *Each node of f is a pair of source and target nodes: $\mathcal{N} \subseteq \mathcal{N}^S \times \mathcal{N}^T$. Let r^S, t^S and r^T, t^T denote the exit and entry nodes of f^S and f^T respectively. Then $r = \langle r^S, r^T \rangle$ and $t = \langle t^S, t^T \rangle$ are the entry and exit nodes of f.*

3. *Each edge of the graph $e = (\langle n^S, n^T \rangle,\ \langle m^S, m^T \rangle) \in \mathcal{E}$, labeled by a pair of instructions $\langle op^S;\ op^T \rangle$, should be justified by one of the following:*

   - *$(n^S, m^S) \in \mathcal{E}^S$ and it is labeled by op^S; $(n^T, m^T) \in \mathcal{E}^T$ and it is labeled by op^T; and op^S and op^T are instructions of the same type (either both reads, writes, assignments, or calls to procedures with the same name).*
   - *$(n^S, m^S) \in \mathcal{E}^S$, labeled by assignment op^S; $n^T = m^T$; and $op^T = \epsilon$.*
   - *$(n^T, m^T) \in \mathcal{E}^T$, labeled by assignment op^T; $n^S = m^S$; and $op^S = \epsilon$.*

   Where, ϵ stands for assignment true?, which represents an idle transition.

Since the edges of a comparison graph are labeled by the same type instructions, reads and writes of the two systems are performed in sync.

A *composed transition* $\langle n;\ \vec{d}\rangle \xrightarrow{e^S;\ e^T} \langle n';\ \vec{d'}\rangle$ is interpreted as a sequential composition of the source and target transitions with one exception. Let e^S and e^T be labeled by $read(\vec{u}^S)$ and $read(\vec{u}^T)$. Then, the transition is enabled only if $\vec{d'}_{u^S} = \vec{d'}_{u^T}$, where $\vec{d'}_{u^S}$ and $\vec{d'}_{u^T}$ are obtained from $\vec{d'}$ by restricting it to the values that correspond to the variables $\vec{u}^S$ and $\vec{u}^T$. Thus, we require that the values fed into the source and target reads are equal. If n is a root node of the procedure, then the transition is enabled only if $\vec{d} = ((\vec{\xi} \circ \vec{\xi}),\ (\vec{\zeta} \circ \vec{\zeta}), \vec{\top})$ for some values $\vec{\xi}$ and $\vec{\zeta}$. In other words, the same values are fed into the input parameters of f^S and f^T. Given σ in $Cmp(f)$, we use $\sigma{\uparrow}_S$ to denote a path obtained by projection of σ onto the states and transitions related to procedure f^S.

Rule 2. *There does not exist σ in $Cmp(f)$ such that $\sigma{\uparrow}_S$ or $\sigma{\uparrow}_T$ contains an infinite sequence of ϵ-transitions.*

The following lemma follows directly from Rule 1 and Rule 2:

Lemma 1. $\forall \sigma \in Cmp(f)\ : (\exists \sigma^S \in Cmp(f^S)\ :\ \sigma^S \sim_{st} \sigma{\uparrow}_S) \wedge (\exists\ \sigma^T \in Cmp(f^T)\ :\ \sigma^T \sim_{st} \sigma{\uparrow}_T)$*; i.e. every computation of the comparison graph has the corresponding computations in both source and target.*

In addition, we should ensure the reverse of the previous claim: the computations of the comparison graph represent all the computations of the input systems. We say that computations of an input system, say $Cmp(f^S)$, are *covered* by $Cmp(f)$ when the following condition holds: $\forall \sigma^S \in Cmp(f^S),\ \exists \sigma \in Cmp(f)\ :\ \sigma^S$ differs from $\sigma{\uparrow}_S$, by only finite sequences of (padding) ϵ-transitions. The notion of coverage is stronger then stuttering equivalence, so it follows that $\sigma^S \sim_{st} \sigma{\uparrow}_S$.

***Rule* 3.** *Computations of f^S and computations of f^T are covered by $Cmp(f)$.*

Note that following Rule 3, not all edges of the input graphs have to be in the comparison graph, which allows us to disregard the unreachable states of the input systems. Now as we have defined a comparison graph, let's consider what properties it should satisfy in order to guarantee the correctness of translation.

Definition 6. *A comparison graph $f = f^S \boxtimes f^T$, is a* witness *of correct translation if for every $((\vec{\xi} \circ \vec{\xi}), (\vec{\zeta} \circ \vec{\zeta}))$-computation of f, its target and source projections have equal observations.* Note, we restrict the computations under consideration to those in which the input parameters are initialized with the same values.

The following theorem shows the relation of the comparison graphs and the correctness of translation.

Theorem 1. *Target function f^T is* a correct translation *of source function f^S if and only if there exists a* witness *comparison graph $f = f^S \boxtimes f^T$. In addition, if f^T is a correct translation of f^S then every comparison graph $f = f^S \boxtimes f^T$ is a witness of correct translation.* (Refer to Section A for the proof of the theorem.)

According to Theorem 1, in order to determine the correctness of translation, it is sufficient to construct a comparison graph and check if it is, indeed, a witness.

Example 2. *Fig. 2.10 depicts an example of a witness comparison graph for procedure $f(\&(Y, y)) = f^S(\&Y) \boxtimes f^T(\&y)$. We use capital variables to denote the variables of the source and their lower case counterparts for the target. Here, the source procedure accepts the argument Y, increments it by 25, reads in an integer*

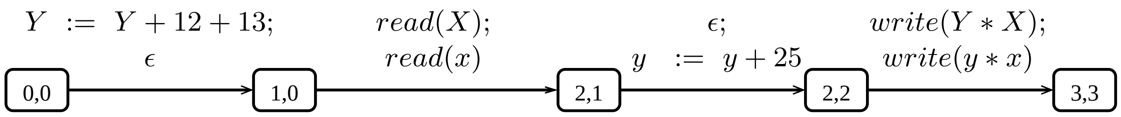


Figure 2.10: A comparison transition graph for $f(\&(Y, y))$

input into X, and outputs $Y * X$. The target procedure performs the same operations; however, the order of the read and the assignment is interchanged due to instruction scheduling. Let σ be the $(\vec{\top}; (5, 5))$-computation of the comparison graph $f(\&(Y, y)) = f^S(\&Y) \boxtimes f^T(\&y)$ defined by the user input 10*. In other words, σ is the computation in which the arguments passed into the source and the target procedures both evaluate to* 5 *and both systems read the value* 10 *form the input device. The observations of the source projection $\sigma{\uparrow}_S$ and the target projection $\sigma{\uparrow}_T$ are equal:*

$$o(\sigma{\uparrow}_S) = o(\sigma{\uparrow}_T) = \perp \xrightarrow{\perp} \perp \xrightarrow{read} (10) \longrightarrow (300) \xrightarrow{write} \perp$$

Witness Verification Conditions

Let φ_n be an assertion associated with a node n. An assertion network $\Phi_{\mathcal{C}} = \{\varphi_n : n \in \textit{locations of } \mathcal{C}\}$ for a program $\mathcal{C}$ is said to be **invariant** if for every state $\langle n; \vec{d}\rangle$ occurring in a computation, $d \models \varphi_n$. That is, on every visit of a computation of node n, the data state satisfies the corresponding assertion φ_n.

Suppose a comparison program $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ and an invariant network have been constructed. The rules presented below can be used to generate `Witness Verification Conditions` for $\mathcal{C}$. Whenever the verification conditions are valid, all the transition graphs that constitute $\mathcal{C}$ are witnesses of correct translation, so we can apply Theorem 1 to safely conclude that the translation is correct; otherwise, we report that the translation is erroneous.

- For a write edge (n, m) labeled by $(\ write(\vec{u}^S);\ write(\vec{u}^T)\ )$:

$$\varphi_n \ \rightarrow\ (\vec{u}^S = \vec{u}^T).$$

- For a call edge $e = (n, m)$ labeled by $(\ g^S(E^S, \vec{u}^S);\ g^T(E^T, \vec{u}^T)\ )$, we check that the call arguments are equal:

$$\varphi_n \ \rightarrow\ (E^S = E^T) \wedge (\vec{u}^S = \vec{u}^T)$$

- If n is the exit node of the comparison transition graph $f^S \boxtimes f^T$, where $f^S(\vec{x}^S;\ \&\vec{z}^S)$ and $f^T(\vec{x}^T;\ \&\vec{z}^T)$, we check if the values of the variables passed by reference are equal:

$$\varphi_n \ \rightarrow\ (\vec{z}^S = \vec{z}^T).$$

Claim 2. *Let $main^S$ and $main^T$ be the main procedures of $\mathcal{S}$ and $\mathcal{T}$ respectively. A comparison graph $main = main^S \boxtimes main^T$ is a witness of correct translation and, consequently, $\mathcal{S}$ is a correct translation of $\mathcal{T}$ if all the* `Witness Verification Conditions` *associated with the comparison program $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ are valid.*

Note that since we are checking that the procedure input parameters are equivalent, the invariant generation algorithm can be intraprocedural. Essentially, one can apply an assume-guarantee reasoning, where f is checked to be a witness, assuming that all the callees of a procedure f are witnesses themselves.

The presented conditions do not constitute an inductive proof of translation correctness: it is assumed that the assertions in $\Phi_{\mathcal{C}}$ are indeed invariants of $\mathcal{C}$. The extra requirement that has to be satisfied in case such a proof is desirable is that the invariant assertion network should be *inductive* [24]. The availability of such a proof increases the level of confidence and allows third-party verification. In addition, it is required if one is to employ invariant generation techniques that may introduce *false positives*, such as probabilistic algorithms [28]. Automatic theorem provers such as Simplify [19], YICES[9], and CVC3[1], can be utilized to independently check the validity of the proof. The notion of inductiveness is formalized in Section 2.2.2.

Next, we describe a method for comparison system construction. However, generation of program invariants and checking their correctness are essential ingredients for solving a translation validation problem. Here, one of the main advantages of our approach comes into play. Since we have reduced the translation validation problem to analysis of a single system, any existing technique out of a vast body of work on invariant generation can be used. From our experiments, we found that, among others, global value numbering [61] and assertion checking – a static program verification technique based on computation of weakest-precondition [20], are quite effective in this setting. Please refer to Section 2.4.5 and Section 2.4.4 for a detailed discussion.

2.4.3 Comparison Graph Construction

The CoVaC framework can be used in various settings. For example, in a setting of a certifying compiler, one can rely on the compiler to build the comparison graph. However, when the compiler internals are unaccessible (or the auxiliary information cannot be trusted), we may have to rely on an algorithm that only requires the two input programs as the input. Note that the compiler hints do not need to be prohibited; indeed, they can be incorporated into the algorithm and simplify the analysis.

In this section, we present such an algorithm for *consonant* input programs (i.e., structurally similar programs) as defined in Section 2.2.4. The restriction to the consonant programs allows for effective application of our methodology to verification of optimizing compilers in absence of compiler annotations. The algorithm discovers the control abstraction and merges the corresponding nodes of the two input systems. To relate the variables of two input systems, our method relies on invariant generation algorithm *InvGen*. Section 2.4.5 gives an overview of one such algorithm; but, in principle, any invariant generation algorithm can be plugged into the CoVaC framework. The second condition of consonance guarantees that such an algorithm is realizable.

Recall that the comparison system $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$ is just a collection of all graphs $f = f^S \boxtimes f^T$, where f^S and f^T are the corresponding procedures from $\mathcal{S}$ and $\mathcal{T}$. Thus, it suffices to present a construction algorithm for a procedure f.

Algorithm `compose`

Fig. 2.11 presents pseudocode for the `compose` algorithm that iteratively constructs a comparison graph $f = f^S \boxtimes f^T$ for consonant input transition graphs f^S and f^T. We start the construction with a node $n_0 = \langle n_0^S,\, n_0^T \rangle$, where n_0^S and n_0^T are the entry nodes of f^S and f^T, respectively. The new node n_0 is added to the `WorkList`, which is our discovery frontier: if a node `n` is placed into `WorkList`, it means that, potentially, more edges outgoing from `n` may be discovered. At each iteration, we remove a node n from the `WorkList` and apply `matchEdges` function to construct a list of newly discovered outgoing edges. The end nodes of the edges, denoted $\mathtt{n}_e$, are placed into `WorkList`[1]. Even though we always discover a new edge, $\mathtt{n}_e$ could have been added to f at some previous iteration and may also have successors in f. In that case, all its eventual successors must also be added to `WorkList`. Intuitively, if a new path leading to a node is added, that node has to be processed again since more outgoing edges could be discovered. The function `matchEdges` may fail, returning `NULL`. This happens when we cannot construct a comparison system satisfying the requirements from Section 2.4.2.

```
//Initialization:
$n_0$:=CompNode($n_0^S$, $n_0^T$); $\mathcal{C}$.Nodes:={$n_0$}; $\mathcal{C}$.Edges:={}; WorkList := {$n_0$};
//Iteration:
while( !  WorkList.isEmpty()) {
   n := WorkList.removeElement();
   MatchList := matchEdges(n,$\mathcal{S}$,$\mathcal{T}$);
   if(MatchList == NULL) ABORT;
   while(!  MatchList.isEmpty()){
      $e_{new}$ := MatchList.removeElement();
      $n_e$ = NewCEdge.toNode();
      $\mathcal{C}$.Nodes.add($n_e$);  //unlike the edge, $n_e$ may not be new[1]
      $\mathcal{C}$.Edges.insert($e_{new}$);
      WorkList.add($n_e$);
      WorkList.add(getDescendants($n_e$));
   }
}
```

Figure 2.11: Algorithm `compose` that constructs $f = f^S \boxtimes f^T$

Below is a set of rules used by `matchEdges` procedure to combine the source and target edges. Every pair of matched edges forms a new edge that is added to the

[1]Procedure *add* does not add duplicate items to a collection.

comparison graph.

- **Matching edges of the same type**: Given a node $\langle n^S, n^T \rangle$, we match the outgoing edges if and only if $\tau(n^S) = \tau(n^T)$. Here, $\tau(n)$ denotes the type of the node n as defined in Section 2.2.4.

- **Adding ϵ–transitions**: If $n^S \notin \mathcal{P}^S$ (implying $\tau(n^S) = ua$), we match the source assignment edge with an ϵ–transition on the target. The case of $n^T \notin \mathcal{P}^T$ is handled analogously.

- **Raising error**: If none of the rules are applicable to a node $\langle n^S, n^T \rangle$, `matchEdges` returns `NULL`, and the construction of $\mathcal{C}$ is aborted.

We always match read, write, and function call edges if both systems can take such a step. Guarded assignment edges can also be composed with each other; but we require that either both systems are currently at a branch node (or a loop head depending on the desired granularity) or neither. Since the input systems are consonant, this condition allows us to align the corresponding source and target cut points. The case when only one of systems has reached a cut point is covered via ϵ–transitions, so that it can wait for the other system to catch up. Note that since ϵ is only composed with unconditional assignments, it is guaranteed that the comparison system does not contain an ϵ–cycle, so the wait always is finite. Finally, we fail when both systems are at cut point nodes n^S, n^T but $\tau(n^S) \neq \tau(n^T)$. For example, one system is ready to read while the other is about to execute a procedure call.

Branch Alignment: Consider the case when $\tau(n^S) = \tau(n^T) = br$. By the first rule of `matchEdges`, the outgoing edges should be matched. However, there is an obvious efficiency problem with simply taking all possibilities (i.e., cartesian product) when we consider two nodes with multiple outgoing assignment edges. Such straightforward approach may lead to a number of edges in f being quadratic in the number of edges in the input graphs. More importantly, if we mismatch the branches, unreachable nodes could be introduced into the graph, which may lead to further misalignment down the road. In particular, read, write, and function call edges may get out of sync. Consider the example in Fig. 2.12. Suppose $C = c$, $X = x$, and $Y = y$. Then f^T is a correct translation of f^S. However, if we compose edges $(0, 1)$ and $(4, 6)$ just relying on the fact that they are both conditional assignments ($\tau(0) = \tau(4) = br$), the algorithm presented so far will raise an error when examining the newly added unreachable node $\langle 1, 6\rangle$. Thus, there is a need for comprehensive branch matching. One such method is presented below; in addition to resolving the misalignment issue, it usually constructs a comparison graph linear in the size of the input graphs.

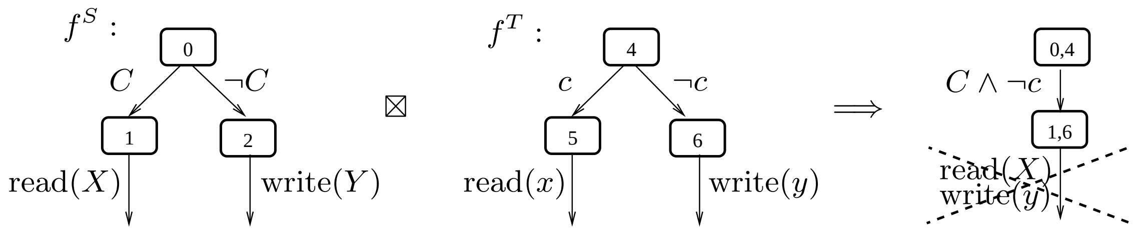


Figure 2.12: Misaligned branches

Assume that we have an algorithm *InvGen*(f_k) that, given a partially constructed graph f_k, obtained after the k^{th} iteration of `compose`, outputs a *set of invariants* $\{\ \varphi_n^k : n \in locations\ of\ f_k\}$. We will use these invariants to facilitate the edge matching at iteration $k+1$ so that the composed edges that would introduce infeasible paths are ruled out. Let $\mathcal{E}_n^S$ represent the set of source edges outgoing from n^S s.t. each edge $e^S \in \mathcal{E}_n^S$ is labeled by $c^S \rightarrow [\vec{u^S} := E^S(\vec{y})]$. We define $\mathcal{E}_n^T$ similarly.

A pair $(e^S, e^T) \in \mathcal{E}_n^S \times \mathcal{E}_n^T$ *is matched if and only if*

– it does not yet belong to the comparison graph and

– $(\varphi_n^k \ \wedge \ c^S \ \wedge \ c^T)$ *is satisfiable.*

We only want to add an edge if there exists an execution through f_k in which e^S and e^T are enabled simultaneously. An important question to ask is how the decision made using a partially constructed graph f_k relates to the fully constructed graph f. Let φ_n^{fix} be the invariant which can be obtained by running *InvGen* on the fully constructed comparison graph f. Invariant φ_n^k is an under-approximation of φ_n^{fix}, meaning, for some assertion Φ_n^k, $\varphi_n^k \ = \ (\varphi_n^{fix} \wedge \Phi_n^k)$.

Lemma 2. *No spurious predictions are possible: if the match* (e^S, e^T) *is made with* φ_n^k*, it also complies with* φ_n^{fix}*.*

Proof. Suppose, a match is made using φ_n^k on iteration $k+1$ and $(\varphi_n^k \ \wedge \ c^S \ \wedge \ c^T)$ is satisfiable. Since $\varphi_n^k \ = \ (\varphi_n^{fix} \wedge \Phi_n^k)$, assertion $(\varphi_n^{fix} \ \wedge \ c^S \ \wedge \ c^T)$ is also satisfiable. Thus, the match would also be made with φ_n^{fix}. □

As a practical consequence of the lemma, algorithm `compose` never has to remove any of the previously added edges; thus, it never backtracks. The converse does not hold: we may discover more matches with invariant φ_n^l, where $l : l > k$ is a later iteration of algorithm `compose`. For this reason, the algorithm adds n_e and its decedents to the `WorkList` (see Fig. 2.11).

Theorem 2. The following are properties of algorithm `compose`:

- *Termination:* algorithm `compose` terminates.
- *Soundness:* if algorithm `compose` succeeds, the resulting graph $f = f^S \boxtimes f^T$ satisfies all of the requirements presented in Section 2.4.2.
- *Completeness:* if f^T and f^S are consonant, `compose` succeeds in construction of a comparison graph $f = f^S \boxtimes f^T$ given a strong enough *InvGen*.

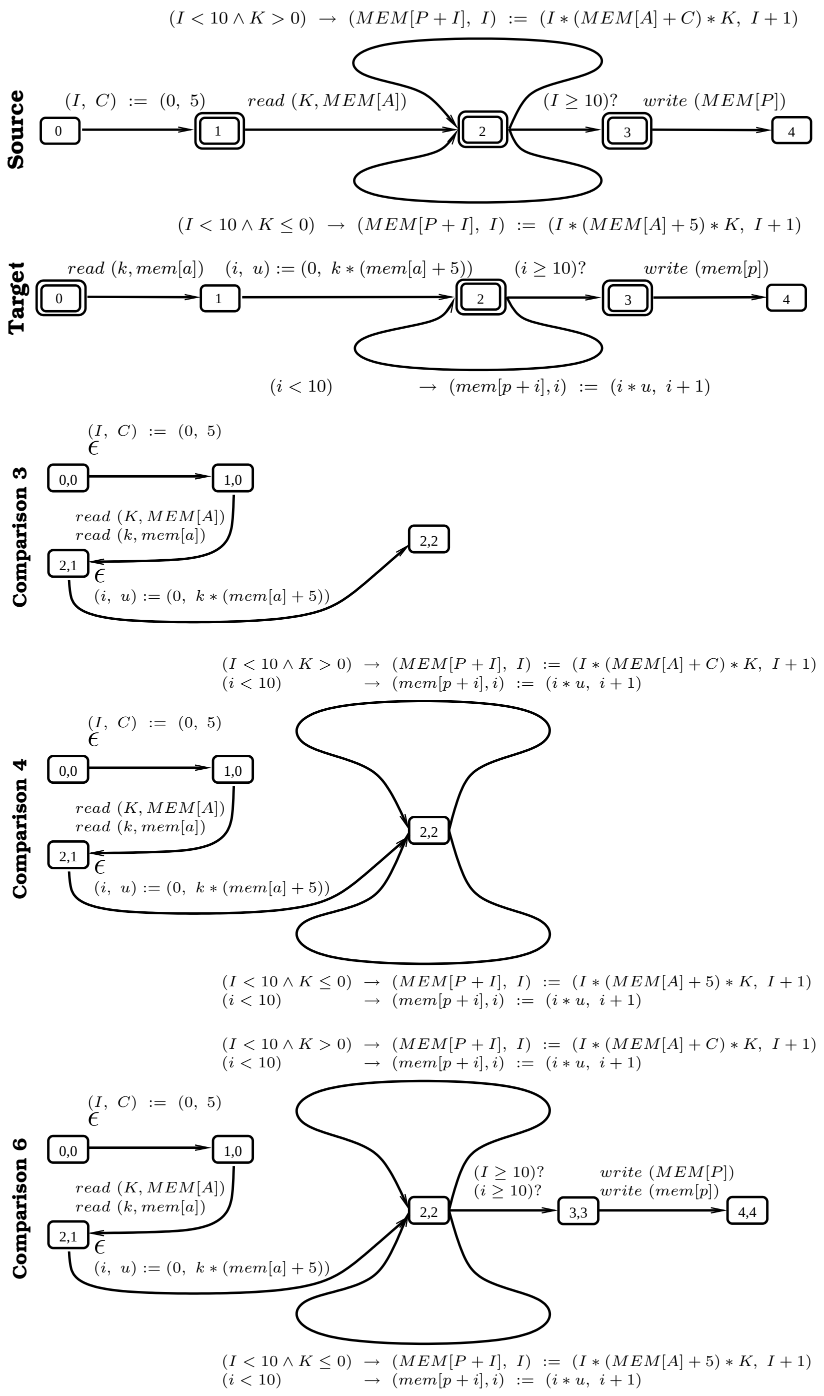
The proof of Theorem 2 is presented in Section A.

Note that the completeness of the algorithm is conditional on strength of *InvGen* algorithm used for branch matching. Even for consonant graphs the invariant may need to be strong enough to express reachability, which is undecidable for infinite state systems. All hope is not lost: it is usually feasible to construct the invariants sufficient for our particular application – verification of compiler transformations, intuitively, due to the fact that compilers base their decisions on automated reasoning. See Section 2.4.5 for the discussion on the *InvGen* used in practice.

Example

In this section, we present an example that demonstrates application of `compose` algorithm to comparison system construction along with the generated invariants and `Witness Verification Conditions`. Consider Fig. 2.13. The first two graphs depict the *source* transition graph and the *target* obtained from the source after constant copy propagation, if simplification, loop invariant code motion, reassociation, and instruction scheduling. Cut point nodes are denoted by double circles. We use capital letters to denote the source variables and their lowercase counterparts for the target. MEM and mem denote the memory heaps. The procedure first reads in two elements – one is stored in register K and the other one in memory at address A. Then, ten elements of the array, stored starting at address P, are assigned to. Finally, the first element of the array is printed out. We assume that the addresses of the array elements do not overlap with A.

Below the source and the target graphs, we depict the comparison graphs obtained at the three stages of the comparison graph construction. After the third iteration of the algorithm `compose` (from Section 2.4.3), we obtain graph $\mathcal{C}_3$ (Comparison 3), which is constructed as follows. On the first iteration, an assignment of the source is matched up with an ϵ-transition on the target. On the second iteration, node $\langle 1, 0\rangle$ is considered, and since both procedures are ready to execute reads, the composed read edge is added. Next, we examine node $\langle 2, 1\rangle$. Since only the source procedure has reached a cut point, it waits for the target system to catch up by taking an ϵ-transition. On the fourth iteration, node $\langle 2, 2\rangle$ is considered for the first time and the algorithm $InvGen(\mathcal{C}_3)$ returns $\varphi^3_{\langle 2,2\rangle}$: $(I = i = 1)$, which



$(I < 10 \wedge K > 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+C) * K,\ I+1)$
Source
$(I,\ C) := (0,\ 5)$
$read\ (K, MEM[A])$
$(I \geq 10)?$
$write\ (MEM[P])$
0
1
2
3
4
$(I < 10 \wedge K \leq 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+5) * K,\ I+1)$
Target
$read\ (k, mem[a])$
$(i,\ u) := (0,\ k * (mem[a]+5))$
$(i \geq 10)?$
$write\ (mem[p])$
0
1
2
3
4
$(i < 10) \rightarrow (mem[p+i], i) := (i * u,\ i+1)$
Comparison 3
$(I,\ C) := (0,\ 5)$
ϵ
0,0
1,0
$read\ (K, MEM[A])$
$read\ (k, mem[a])$
2,1
2,2
ϵ
$(i,\ u) := (0,\ k * (mem[a]+5))$
$(I < 10 \wedge K > 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+C) * K,\ I+1)$
$(i < 10) \rightarrow (mem[p+i], i) := (i * u,\ i+1)$
Comparison 4
$(I,\ C) := (0,\ 5)$
ϵ
0,0
1,0
$read\ (K, MEM[A])$
$read\ (k, mem[a])$
2,1
2,2
ϵ
$(i,\ u) := (0,\ k * (mem[a]+5))$
$(I < 10 \wedge K \leq 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+5) * K,\ I+1)$
$(i < 10) \rightarrow (mem[p+i], i) := (i * u,\ i+1)$
$(I < 10 \wedge K > 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+C) * K,\ I+1)$
$(i < 10) \rightarrow (mem[p+i], i) := (i * u,\ i+1)$
Comparison 6
$(I,\ C) := (0,\ 5)$
ϵ
0,0
1,0
$read\ (K, MEM[A])$
$read\ (k, mem[a])$
2,1
2,2
$(I \geq 10)?$
$(i \geq 10)?$
3,3
$write\ (MEM[P])$
$write\ (mem[p])$
4,4
ϵ
$(i,\ u) := (0,\ k * (mem[a]+5))$
$(I < 10 \wedge K \leq 0) \rightarrow (MEM[P+I],\ I) := (I * (MEM[A]+5) * K,\ I+1)$
$(i < 10) \rightarrow (mem[p+i], i) := (i * u,\ i+1)$

Figure 2.13: CoVaC example: the source, the target, and the comparison graphs

is used to align the branches of the loop and obtain $\mathcal{C}^4$ (Comparison 4). However, $\varphi^3_{\langle 2,2\rangle} \;\wedge\; (I \geq 10) \;\wedge\; (i \geq 10)$ is unsatisfiable. Thus, the matching of the loop exit edges is ruled out by the invariant. At the end of the fourth iteration, node $\langle 2, 2\rangle$ is added to the `WorkList` again. Notice that $\varphi^3_{\langle 2,2\rangle}$ does not hold in system $\mathcal{C}^4$ since I and i are updated in the loop, so $\mathit{InvGen}(\mathcal{C}^4)$ widens the invariant, resulting in $\varphi^4_{\langle 2,2\rangle} \;:\; (I = i)$, which allows to match up the loop exit edges ($\varphi^4_{\langle 2,2\rangle} \;\wedge\; (I \geq 10) \;\wedge\; (i \geq 10)$ is satisfiable). Finally, we match the write edges and obtain $\mathcal{C} = \mathcal{C}_6$ (Comparison 6).

After the comparison system is constructed, we generate the `Witness Verification Condition` to check that both systems write out the same values (following the rules from Section 2.4.2):

$$\varphi^{\mathit{fix}}_{\langle 3,3\rangle} \rightarrow (\mathit{MEM}[P] = \mathit{mem}[p]).$$

An *inductive* invariant network for the comparison program is presented below. The validity of the verification condition directly follows from $\varphi^{\mathit{fix}}_{\langle 3,3\rangle}$.

$$\begin{aligned}
\varphi^{\mathit{fix}}_{\langle 0,0\rangle} &: (\mathit{MEM} = \mathit{mem}) \;\wedge\; (A = a) \;\wedge\; (P = p) \;\wedge\; (A \notin [P..P+9]) \\
\varphi^{\mathit{fix}}_{\langle 1,0\rangle} &: (I = 1) \;\wedge\; (C = 5) \;\wedge\; \varphi^{\mathit{fix}}_{\langle 0,0\rangle} \\
\varphi^{\mathit{fix}}_{\langle 2,1\rangle} &: (K = k) \;\wedge\; \varphi^{\mathit{fix}}_{\langle 1,0\rangle} \\
\varphi^{\mathit{fix}}_{\langle 2,2\rangle} &: (I = i) \;\wedge\; (u = (\mathit{MEM}[A] + C) * K) \;\wedge\; (C = 5) \;\wedge\; \varphi^{\mathit{fix}}_{\langle 0,0\rangle} \\
\varphi^{\mathit{fix}}_{\langle 3,3\rangle} &: (\mathit{MEM} = \mathit{mem}) \;\wedge\; (P = p)
\end{aligned}$$

$\varphi^{\mathit{fix}}_{\langle 0,0\rangle}$ asserts our assumptions that at the entry to the programs, the memory heaps of the source and target are the same; the corresponding address variables of the two systems are equal; and the address variable $A(a)$ does not overlap with the addresses of the elements of the source(target) array. The most interesting invariant

is $\varphi^{fix}_{\langle 2,2\rangle}$, which asserts that, after each loop iteration, the source and target heaps stay equivalent. Its validity is ensured by the following facts: the addresses at which memory is updated are equal (due to $I = i \ \wedge \ P = p$); the expressions stored at those addresses are equal (since $u = (MEM[A] + C) * K$
$\wedge \ I = i$); $MEM[A]$ is not altered by the loop (because $A \notin [P..P + 10]$).

2.4.4 Correlating the Unbounded Heaps

Dynamic memory allocation is a very powerful and popular programming paradigm. However, along with its power, comes the complexity that is often difficult to analyze. In the CoVaC setting, the comparison system contains two unbounded memory regions from which the memory can be allocated and whose contents are manipulated. This section is devoted to a program analysis technique that can be used to generate the program invariants that show how these two memory regions (or heaps) are related.

Program heaps are modeled by unbounded arrays, which allows us to employ the theory of arrays available in theorem provers. For example in CVC3 [1], a heap would be modeled by `ARRAY INT OF REAL`.

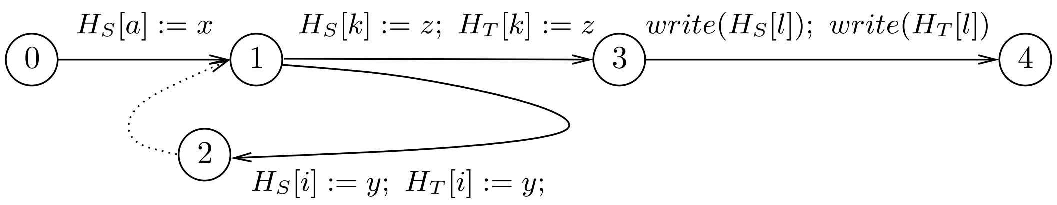

$H_S[a] := x$
$H_S[k] := z; \ H_T[k] := z$
$write(H_S[l]); \ write(H_T[l])$
0
1
3
4
2
$H_S[i] := y; \ H_T[i] := y;$

Figure 2.14: Heaps equivalence example

Consider the comparison system example in Fig. 2.14. Here, H_S and H_T denote the heaps of the source and the target programs. For simplification purposes, only operations that involve H_S and H_T are shown in the figure. In addition, source and target operations share the same variables as a result of value numbering. We assume that a and b are aliases. Since $H_S[b]$ is being assigned to by the edge $(1, 3)$, the assignment to the source heap $H_S[a] := x$ is redundant and is removed in the target. The assignment $H_S[k] := i$ is also redundant since k is not updated within the loop and the value of $H_S[k]$ is altered by the edge $(1, 3)$. In order to determine if the constructed graph is a witness, the assertion checker needs to determine if the values at the corresponding heap locations are equal: $(H_S[l] = H_T[l])$, for some address l. Since the heaps are dynamically updated within the loop, the number of locations which have to be considered can be unbounded. In addition, due to various optimizations like code motion and dead code elimination, the source and target heaps are not equal to each other at each node of the comparison program. Going back to our example, since the redundant stores $H_S[a] := x$ and $H_S[k] := i$ are eliminated in the target program, H_S may not be equal to H_T at the graph nodes 1 and 2. The key assumption we use is that, at each node n of the graph, the heaps only differ from each other at a finite set of memory locations and the values at the rest of heap locations are equivalent. This assumption is fair in a setting of compiler validation.

For our analysis, we assume that the input comparison system is in SSA form [17]. Let $\mathcal{N}_\mathcal{C}$ denote the set of nodes of the comparison graph $\mathcal{C}$ (which can be either a partially or a fully constructed comparison graph). Next, we describe the procedure that computes $\Delta_n : n \in \mathcal{N}_\mathcal{C}$ - the set of symbolic heap locations at which the heaps may possibly differ.

For every node n, Δ_n is initialized with $\emptyset$. Then, we iterate and at each iteration update the deltas according to the equation below. We stop when there is no change. In other words, the set of deltas is computed as the minimal fixed point of the equation.

Data Flow Equation: Let E_n be the set of edges incoming into node n. For an edge $e \in E_n$, let $head(e)$ denote the head node of e; and let δ_e denote the set of heap locations that have been updated by the instructions of e.

$$\boxed{\Delta_n \ := \ \Delta_n \bigcup \ reduce(\bigcup_{e \in E_n} \{\Delta_{head(e)} \bigcup \delta_e\}, \ n \ )}$$

For every edge e incoming into n, we add to the set Δ_n the locations at which the heaps may differ prior to executing the instructions of e and the locations that have been updated by e. Note that e may update H_S and H_T by storing the same expression at a location l. In that case, the $H_S[l] = H_T[l]$ at n and Δ_n should not include l. The *reduce* procedure removes the locations at which the heaps will become equal once we arrive at location n:

```
$reduce \ ( \ SymbolicLocationsSet \ X_n, \ Node \ n \ )$
  $for \ each \ l \in X_n :$
    $if \ ( \ check\_assertion(H_S[l] = H_T[l], \ n) \ )$
      $X_n = (X_n \setminus l);$
  $return \ X_n;$
```

In the pseudocode above, we use the assertion checker to determine if the values stored at the two heap locations are equal at node n. The assertion checker uses the invariants based on alias analysis and the invariants generated from the $\Delta_i, \ i \in \mathcal{N_C}$ computed at the previous iteration. The invariant generation is described below.

An additional check has to be performed if the edge $e = (m, n)$ is a loop back edge. If any address from the set $\Delta_m \backslash \Delta_n$ is modified in the loop (a possibly different heap location is modified on each iteration of the loop), we report an error - the number of locations at which the heaps differ may be unbounded.

Invariant Generation: Given the computed Δ_n, we generate the following invariant for a node n:

$$\boxed{\varphi_n = \forall i \in \mathbf{Z}\ (\bigwedge_{\forall l \in \Delta_n} i \neq l \quad \rightarrow \quad H_S[i] = H_T[i]\ )}$$

Going back to the example from Fig. 2.14, Table 2.1 displays the delta sets generated after each iteration.

	Δ_0	Δ_1	Δ_2	Δ_3	Δ_4
Initialization	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
Iteration 1	$\emptyset$	$\{a\}$	$\{k\}$	$\emptyset$	$\emptyset$
Iteration 2	$\emptyset$	$\{a,\ k\}$	$\{a,\ k\}$	$\emptyset$	$\emptyset$
Iteration 3	$\emptyset$	$\{a,\ k\}$	$\{a,\ k\}$	$\emptyset$	$\emptyset$

Table 2.1: The heap delta sets

Let's consider the second iteration of the algorithm. When considering node 1, $\Delta_1 = \{a\} \bigcup reduce(\delta_{(0,1)} \bigcup \Delta_2,\ 1) = \{a\} \bigcup reduce(\{a,\ k\},\ 1) = \{a,\ k\}$. Next, node 2 is processed and we compute $\Delta_2 = \{k\} \bigcup reduce(\ \Delta_1 \bigcup \delta_{(1,2)},\ 2) = \{k\} \bigcup reduce(\ \{a, i, k\},\ 2) = \{a, k\}$. Since the edge $(2, 1)$ is a back edge, we check that k is not updated within the loop. When node 3 is processed, $\Delta_3 = reduce(\ \Delta_1 \bigcup \delta_{(1,3)},\ 3) = reduce(\{a, k, b\},\ 3) = \emptyset$. All the locations are removed by *reduce* since a and b are aliases, and the source and target heaps are overwritten with the same values at k and b. Finally, we compute $\Delta_4 = reduce(\Delta_3,\ 4) = \emptyset$. The computation stabilizes after three iterations.

The corresponding invariants can be encoded as the following predicates in CVC3:

$$\varphi_0 = \varphi_3 = \varphi_4 : \quad FORALL\ (i : INT) : (H_S[i] = H_T[i])$$

$$\varphi_1 = \varphi_2 : \quad FORALL\ (i : INT) : ((i \neq a) \& (i \neq k))$$
$$=> (H_S[i] = H_T[i])$$

Below is a more efficient version, which can also be used if the theorem prover does not support quantification:

$$\varphi_0 = \varphi_3 = \varphi_4 : \quad Hs = Ht$$

$$\varphi_1 = \varphi_2 : \quad ((H_S\ WITH\ [a] := H_T[a])$$
$$WITH\ [k] := H_T[k]) = Ht$$

Claim 3. *If the algorithm terminates without an error, for every $n \in \mathcal{N}_\mathcal{C}$, the generated φ_n is invariant at n.*

Proof. Assume that is not the case. Let the path π from the procedure entry r to some node n be a shortest counter-example. Then, there exists a heap location i, such that $H_S[i] \neq H_T[i]$ at n, while φ_n asserts otherwise. Meaning, there is no symbolic location $l \in \Delta_n$ that evaluates to i at n.

Consider the last time node n is processed. Suppose, the edge (v, w) is the last edge on the path π that assigned to the heap at location i. Then, there is a location $l \in \delta_{(v,w)}$ that evaluates to i. The location l will be propagated to n according to the data flow equation, unless it is reduced or the value of l is changed by a loop (the second would lead to an early termination with an error). Let's show that l cannot be reduced and thus belongs to Δ_n. Assume wrongly that *check_assertion*$(H_S[l] = H_T[l], u)$ returns *true* for some node u along the path from w to n. However, since $H_S[l] \neq H_T[l]$ at u for the execution π, it must be that one of the invariants associated with the nodes appearing on π from the beginning up to the last occurrence of u, but not including u, does not hold. Therefore, the counterexample π can be truncated starting from u, resulting in a shorter counterexample, which is a contradiction.

To finish the proof, we just need to show that l must evaluate to i at the last state of π. Note that l evaluates to i when we were taking the edge (v, w). The value is unchanged since the procedure is in SSA form and l is not being assigned in a loop.

□

Claim 4. *The algorithm terminates.*

Proof. Termination is guaranteed since the number of locations added to $\Delta_n : n \in \mathcal{N}_\mathcal{C}$ monotonically grows; and the number of symbolic locations is limited by the number of program expressions. □

The number of iterations is bounded by $\mathcal{N}_\mathcal{C} * c$, where c is the number of heap assignments. In practice, we rarely need to iterate for that long. First, we process the nodes in the topological order and use the most recently computed deltas, instead of the results obtained at the previous iteration. In addition, since loops usually have zero net effect on delta, it is uncommon that a node is processed more than twice.

Sound Treatment of Procedure Calls: Our analysis is intraprocedural. To ensure soundness, we check for the following:

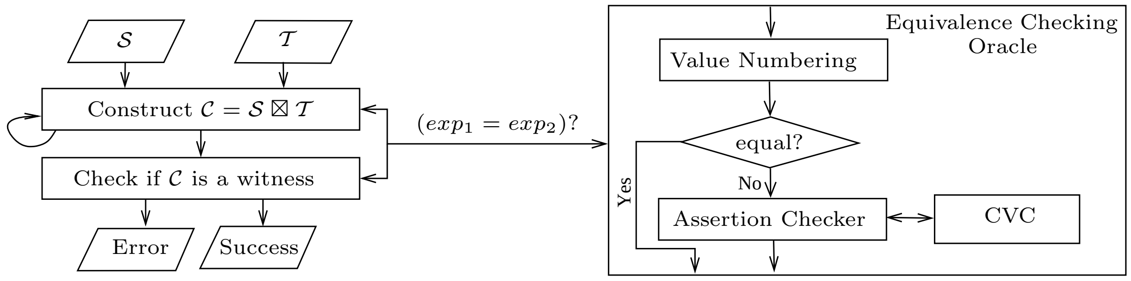
- If an edge from a node n to a node m is a call to procedure foo, the procedure foo must not access the heaps at the locations in Δ_n. In fact, when dealing with compiler verification, either Δ_n is an empty set, or simple alias analysis are sufficient to check the condition.

- If a node r is the procedure exit node, $\Delta_r = \emptyset$. For the entry node t, it is assumed that $\Delta_t = \emptyset$ (Recall that the Δ_t is initialized with $\emptyset$ and is never updated by the algorithm since the entry node does not admit any input edges). This condition ensures the zero net effect of the procedure. Consequently, for a call edge e, $\delta_e = \emptyset$.

2.4.5 CoVaC Implementation

We have constructed a prototype CoVaC tool based on the presented techniques and used it to verify the optimizations performed by LLVM compiler. LLVM [7] is an open source compiler for C and C++. The tool has been developed in C++ and uses LLVM data structures for program representation and parsing. Its current line count is at approximately 7,000. We currently support a subset of C, which does not include function pointers, variable argument function calls, and support for system level calls such as threads and long jumps. We also assume that the types of the variables are either unbounded integers, unbounded reals, or aggregates of the two base data types. The overall work flow of the CoVaC tool is presented in Fig. 2.15.

First, we transform the source and target input procedures into transition graph representation. Placing one node per each loop gives a minimally necessary set (see Section 2.2.1). Model construction based on a minimal set accommodates a wide set of optimizations, but it may also cause an exponential blow-up in the size of the model since we may need to enumerate all possible paths between the



$\mathcal{S}$
$\mathcal{T}$
Construct $\mathcal{C} = \mathcal{S} \boxtimes \mathcal{T}$
Check if $\mathcal{C}$ is a witness
Error
Success
$(exp_1 = exp_2)$?
Equivalence Checking Oracle
Value Numbering
equal?
Yes
No
Assertion Checker
CVC

Figure 2.15: The work flow of the CoVaC tool

nodes. Specifically, if a path from one node to another in the original program has k consecutive if statements, that would amount to 2^k edges in the the transition graph. Another choice is to pick a location before each branch as a node, which guarantees a linear-size model. A deeper analysis (one node per loop head) should be performed only if the lightweight phase fails to prove translation. Note, such failure is rare - it occurs mainly when optimizations change the order of conditional branches. Presently, the CoVaC tool supports only the lightweight phase, where each branch is represented by a node.

After the transition graphs are constructed, CoVaC runs the `compose` algorithm (from Section 2.4.3) to produce a comparison system and then generates the `Witness Verification Conditions` to verify the correctness of translation (as described at the end of Section 2.4.2). If it succeeds in building a comparison graph and the verification conditions are valid the tool asserts `Success`. Otherwise, it raises an `Error`, which signifies that either an error in translation is detected or we ran into a transformation that is not currently supported.

The completeness and efficiency of the CoVaC approach heavily depends on invariant generation algorithms. The framework relies on auxiliary invariants to generate the comparison graph and to check if a generated graph is a witness of correct translation. We follow two strategies to obtain a practical solution. First, the assertions that are generated are goal oriented. In particular, we assume that we only need to check for the validity of the formulas of the form $exp_1 = exp_2$. Second, we utilize a two-phase strategy where each phase provides a certain balance of precision and efficiency. In the first phase, we apply fast lightweight analysis. When it is not sufficient, we resort to deep and precise analysis.

Invariant Generation via Equivalence Checking

As shown in Fig. 2.15, instead of a general purpose invariant generation algorithm, CoVaC tool uses an oracle that checks if two input expressions are equivalent at a particular program location. Checking two expressions for equivalence is sufficient when confirming whether a graph is a witness of correct translation. We just need to ensure that at every node preceding the write instruction, the values that are being printed by the source and the target are the same. Another place where we need auxiliary invariants is branch alignment. We optimize the general approach and align branches by checking equivalence of the corresponding conditions instead of checking the satisfiability of the conjunctions as described in Section 2.4.3. While this approach is less precise, it is still powerful enough to handle most classical compiler optimizations.

Each time we have to align the conditional assignments, we essentially match a branch instruction (or an if-statement) on the source with the one on the target. Assume the source edge e^S_+ is taken when C^S holds; and e^S_- is taken when $\neg\ C^S$ holds. Similarly, there are two edges on the target: e^T_+ and e^T_-, which are conditioned on c^T. Instead of checking the four formulas for satisfiability (following the method in Section 2.4.3), we use the fact that we are dealing with branch instructions, where the conditions are negations of each other, and consider the following cases:

- $(C^S \Leftrightarrow c^T)$ is valid - the conditions are equal; thus, the following edges are matched: $(e^S_+,\ e^T_+)$ and $(e^S_-,\ e^T_-)$.
- $(C^S \Leftrightarrow \neg\ c^T)$ is valid - one condition is the negation of the other; the following edges are matched: $(e^S_+,\ e^T_-)$ and $(e^S_-,\ e^T_+)$.
- Otherwise, we assume that the conditions are not related, so either all possible matches have to be made:
  (e^S_+, e^T_+), $(e^S_-,\ e^T_-)$, $(e^S_+,\ e^T_-)$, and $(e^S_-,\ e^T_+)$, or we can use an ϵ-transition and freeze the execution of the source system, obtaining the following matches: $(e^S_+,\ \epsilon)$; $(e^S_-,\ \epsilon)$. The second option turns out to be better suitable in practice. It corresponds to an optimization in which both branches of a source *if*-statement are removed by an optimizer.

The only case that we have not yet considered is when the conditions overlap. For example, $C^S = (x \geq 5)$ and $c^T = (x \geq 6)$. In this case, the set of the edges should be $(e^S_+,\ e^T_+)$; $(e^S_+,\ e^T_-)$; $(e^S_-,\ e^T_-)$. We would have to use the general matching rule to determine this dependency. However, the checks for such overlaps are rarely needed when dealing with proving translation in presence of compiler optimizations. The only exception is when one of the branches of a source if-statement is removed due to branch simplification. To address this optimization, we execute a pre-processing phase on both input programs in which we simplify the conditionals that evaluate to *true*.

In order to cover most common compiler optimizations, the algorithm has to reason using abstractions of uninterpreted functions and linear arithmetic. The problem of checking equality assertions in programs abstracted in the combined theory of linear arithmetic and uninterpreted functions, and whose conditionals are treated as non-deterministic, is coNP-hard [29]. Nevertheless, there exist efficient methods that are useful in determining the relation between source and target expressions.

As depicted in Fig. 2.15, we employ a value numbering algorithm first. Global value numbering [47] assigns the same value number to provably equivalent variables and expressions throughout the procedure. This technique is particularly effective since we need an oracle to decide the validity of the formula $exp_1 = exp_2$. Value numbering is both fast and capable of detecting many value matches between the source and target expressions, especially, in the code fragments that have not been heavily optimized. Note that even when no optimizations are applied and the input systems are identical up to the renaming of the variables, there must be a technique in place capable of efficiently determining if the corresponding two variables are equal. We use the algorithm by Simpson [61], which provides a good balance between reasoning in theories of uninterpreted functions and linear arithmetic: it can detect a vast majority of equalities of expressions whose operators are treated as uninterpreted functions but also can easily handle simple constant folding and algebraic identities.

When value numbering is not strong enough to determine if two expressions are equivalent (due to excessive optimization), we resort to assertion checking - a static program verification technique based on computation of a weakest-precondition [20]. We generally follow methods like the one described in [12], for development of our assertion checker. A typical assertion checker (or a static program verifier) takes as an input a program and some assertion and generates from these a verification condition that implies the validity of the assertion in the program. The validity of the verification condition is checked by a theorem prover. We use CVC3 [1], an automatic theorem prover for the Satisfiability Modulo Theories, as a back end validity checker. Uninterpreted functions are used to represent the operators that are not supported by CVC3. As long as the compiler does not perform any simplifications based on the semantics of these operators, there is no loss of precision. The negative result - the expressions are not equal, is reported if we are unable to determine if two expressions are equivalent (for example, when the theorem prover is not strong enough to determine the validity of a verification condition). This ensures soundness of the method.

As a preprocessing step to the assertion checker, we simplify the input procedure based on the results of the value numbering: the same name is used to represent the variables with the same value number. This turns out to be crucial for both precision and efficiency of the assertion checker. Additional loop invariants, similar to those used in general purpose static verifiers [60], are used by the assertion checker. In addition to using the existing invariant generation techniques, we have developed the novel method that we use for proving equivalence of unbounded heaps. It is described in Section 2.4.4.

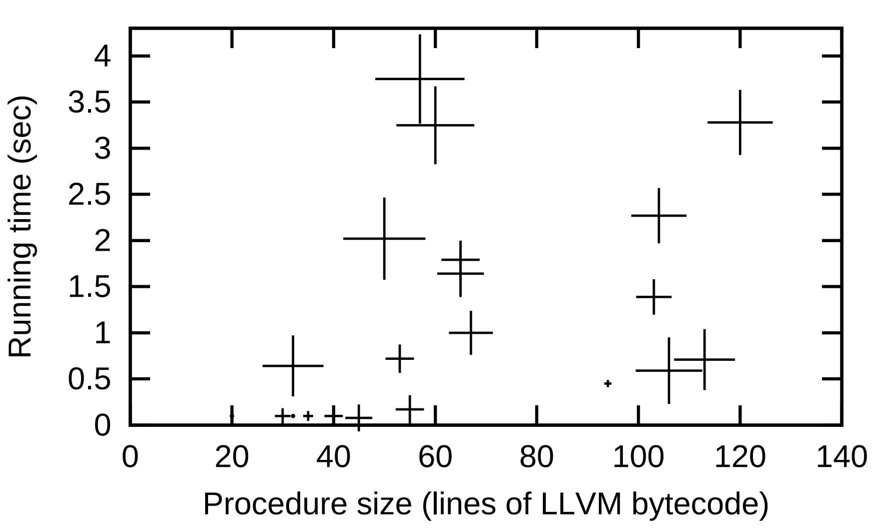


Figure 2.16: Dependency between the running time and the size of the procedure

2.4.6 Experimental Results

We have constructed a prototype CoVaC tool based on the presented techniques and used it to verify the optimizations performed by LLVM compiler. The tool has been tested on a set of procedures with the total line count of 2K of LLVM bytecode, compiled from the selected LLVM and CoVaC feature tests and third party implementation of the classical algorithms like binary search, in-place heapsort, mergesort, Qsort, strcmp, primality testing, shortest paths, etc. All the presented benchmarks contain dynamically allocated data structures; thus, they require the heap equivalence analysis presented in Section 2.4.4. On average, when validating highly optimized code (1/2 optimizations per line), CoVaC spends 0.02 seconds per every line of code. Fig. 2.16 shows the dependency of CoVaC tool running time on the procedure size. The size of the 'cross' is proportional to the number of optimizations performed. The most time is spent on assertion checking, which is dispatched once per every 8 lines when it is difficult to find a strong invariant with value numbering alone. This explains why the dependency of running time on the procedure size and the number of optimizations is not always consistent. We believe that the prototype's performance provides a strong evidence that a practical validator can be constructed (note that unlike a compiler, the tool is used few times per program's lifetime).

2.5 Related Work

We have discussed the different approaches to compiler validation in Section 2.1 and refer the reader to [18] for a survey of compiler verification literature. This section is devoted to translation validation, which is the idea behind the two presented frameworks: ITV and CoVaC.

Translation validation approach was first introduced in [53] for verification of translations from SIGNAL to C. Taking into account the difficulty of proving the correctness of a particular translator or compiler, the authors proposed to tackle a different problem. They have developed an automatic technique for verifying that a particular translation (or a compiler run) produces the correct target code. The approach is based on establishing an abstraction mapping (or refinement mapping) and an inductive proof of it's correctness. The described method is particularly suitable for programs with a restricted control flow structure. In particular, the C-code resulting from the SIGNAL to C translation consists of a single main loop whose body is a loop free program.

In [51], the idea of translation validation was successfully applied to verification of optimizing transformations performed by the GNU C optimizing compiler. The verifier has been tested on very large code bases with the false alarm rate of about 10%. The method is based on inferring and then proving a simulation relation between two programs $\mathcal{S}$ and $\mathcal{T}$ (the source and the target). The proposed algorithm first analyzes the source and target programs and generates a set of symbolic constraints. The major component of this phase is a branch navigation module, which uses simple heuristics and compiler annotations to match structure of the control flow graphs of the source and the target programs. The generated symbolic constraints are solved during the second phase of the algorithm. The set of the transformations it can handle is limited to intraprocedural structure preserving optimizations.

TVOC [67, 68, 39, 69, 21] is a tool for validation of structure preserving optimizations and loop reordering optimizations performed by the ORC compiler [2]. As described in Section 2.3, rule VALIDATE is used for verification of structure preserving optimizations. It is based on the computational induction approach [24], first introduced for establishing properties of a single system. The TVOC tool relies on the compiler annotations to provide the control abstraction and an under-approximation of the data abstraction. Such annotations are often generated by the compilers to facilitate debugging. However, they are not guaranteed, especially at the high optimization levels and when the compiler under consideration is in the development stage. In order to refine the data abstraction, the invariants have to be generated ahead of time, which requires them to be as general as possible. The construction of the data abstraction involves the maximal fixpoint computation that is expensive since, at each iteration, it submits a query to the validity checker. Rules for loop reordering transformations can be applied in addition to the VALIDATE rule to verify transformations such as loop interchange, fusion, distribution, and tiling. The framework has to be extended to incorporate language features such as dynamic memory allocation and interprocedural optimizations.

While the previously described validators handle only transformations over the same intermediate languages, [59] relates the C source code with the generated PowerPC assembly code. The validator uses symbolic transfer functions to represent the behavior of code fragments. It also relates the notions of translation validation and invariant translation, where a concrete semantic interpretation of the symbolic transfer functions corresponds to translation validation while an abstract semantic interpretation corresponds to invariant translation. The approach leads to creation of a unifying framework for the certification of compilation and of compiled programs, which has been applied to verification of non-optimizing transformations of the GCC compiler [6]. Like TVOC, the framework depends on the compiler debug information to relate the source and target variables and program locations.

To facilitate the checking, both [68] and [59] generate invariants over the variables of the source system based on the existing program analysis (like alias analysis) and specially developed techniques [22].

The described approaches are targeted at a wide set of source code transformations. Moreover, translation validation has been applied to creation of verification procedures targeted at specific optimizations. The advantage of the specialized validators is that they are complete (raise no false alarms) and more efficient. [40] describes a complete method for translation validation of register allocation and spilling. The algorithm relies on data-flow analysis, which computes and relates the webs of def-use chains. The tool can also produce detailed explanations of errors. While several of the papers mentioned above come with on paper proofs, none has been mechanically verified. A formally verified translation validator of instruction scheduling optimizations is presented in [62]. Further, [64] presents a mechanically checked proof of correctness of a validation algorithm for a lazy code motion. Both proofs were done using the Coq proof assistant [5] and the validators were integrated in the CompCert [4] formally verified compiler.

2.6 Conclusions

We presented two frameworks for compiler validation - ITV and CoVaC.

ITV is a novel framework for automatic translation validation of reactive programs in presence of interprocedural optimizations. In order to prove the context sensitive constant copy propagation, the framework relies on a special algorithm for invariant generation. ITV is effective when one has access to the mappings between the variables and locations of the source and target programs.

CoVaC is a framework for checking program equivalence based on construction of a cross-product system, which reduces the program equivalence problem to verification of a single program and allows for utilization of the existing off-the-shelf program analyses and tools. In particular, we have shown how the CoVaC framework can be applied to verification of non-cooperative compilers and used it on practice to validate a wide range of optimizations performed by an aggressive modern compiler, LLVM [7]. In addition, we presented a novel invariant generation algorithm useful for proving equivalence of two unbounded memory regions, which is necessary when the input programs contain dynamically allocated data structures. We showed how this and the other existing program analysis techniques can be plugged into the CoVaC framework.

Since we apply the translation validation approach to verification of infinite state systems, there is no hope to have a complete method for proving correct translation in general. The program equivalence problem is undecidable and weaknesses in the analysis lead to false alarms. The tools may report and error even in case the translation is correct. However, because the focus is only on compiler optimizations, the number of false alarms can be drastically minimized or even eliminated. Intuitively, since we are aware of the analysis used by the optimizing compilers, we are optimistic in creation of a strong enough set of auxiliary invariants.

Many interesting questions remain. For example, CoVaC has yet to be extended to support interprocedural optimizations. We are also interested in investigating application of the CoVaC framework to development of a self-certifying compiler and validation of language-based security properties, specifically, checking conformance with information flow policies [13]. Finally, since we are ultimately interested in correcting the bugs, it is essential to explore the ways in which the validator failures can be analyzed and used to pinpoint compilation errors.

Chapter 3

Verifying multithreaded C programs with pancam

3.1 Introduction

The compiler verification methods proposed in Chapter 2 assume that the code of the source program satisfies the desired properties. However, even well understood protocols such as Peterson's protocol for mutual exclusion, whose algorithmic description takes only half a page, have published implementations that are erroneous. Verification of implementation code, as opposed to checking abstract algorithmic descriptions is hard. This is especially the case for programs written in the C language, which has several constructs (such as function pointers, pointer arithmetic and arbitrary type casting) that are difficult to model faithfully. And it is an unfortunate fact of life that the programs most in need of verification are those that use such constructs the most, viz., C programs for embedded systems. Further, the ITV and

CoVaC frameworks mainly focus on the optimization pass of the compiler. Consequently, the compiler verification methods only cover a small part on the path from a design to an executable. The assumption is that the rest of the translation is verified elsewhere.

Instead of proving the properties of a model and then ensuring that they are preserved by implementation and compilation, we propose to model check the programs directly at the level of optimized bytecode. In this chapter, we present *pancam* – a model checker for optimized LLVM [7] bytecode. There are several advantages in analyzing the bytecode as opposed to the executable. A native executable can be difficult to decode. For example, when variable-width instructions are allowed (like in Intel x86), the meaning of a program changes depending upon where decoding begins. More importantly, even the decoded binaries lack the type information, which is essential for program analysis.

pancam can be thought of as a specialized interpreter. It first compiles a multi-threaded C program into optimized LLVM bytecode format. The framework relies on Spin [8], an existing explicit state model checker, to orchestrate the program's state space search and utilizes the bytecode interpreter to compute the program transitions and states. The advantage of using Spin is that we get the functionality of the robust tool that has been in development for almost 30 years for free. Our interpreter is compatible with most of Spin's search options and optimization flags, such as bitstate hashing and multi-core checking. pancam provides support for dynamic memory allocation (the malloc and free family of functions) and for the pthread library, which provides primitives often used by multi-threaded C programs.

When model checking the C implementations is the goal, scalability is the main concern. Since the state space of even a small C program is typically much larger than that of most algorithmic models, we consider various approaches to combat the space explosion problem. In particular, we have developed a partial order reduction method that reduces the number of context switches using dynamic knowledge computed on-the-fly, while being sound for both safety and liveness properties. We also discuss how context-bounding [48] can easily be integrated within our tool, with only a small modification to the interpreter. pancam users can constrain the state space even further by defining the data abstraction functions.

The rest of this chapter is organized as following. We give an overview of the Spin model checker and its support for embedded C code in Section 3.2. Next, Section 3.3 explains how Spin and pancam are integrated. pancam supports abstraction and context-bounded checking as described in Section 3.4. Section 3.5 is devoted to the superstep partial order reduction technique. We present the experimental results in Section 3.6 and review the related work in Section 3.7.

3.2 Background

3.2.1 Spin

Spin [8, 34] is an open-source explicit state model checker for distributed software systems. It has been in development for almost 30 years and has been recently recognized by the Association for Computing Machinery with the prestigious System Software Award for 2001. Spin is targeted at verification of high level models of concurrent systems. It uses specialized PROMELA language for model capture and fully supports Linear Temporal Logic (LTL) [54] as its property specification language.

The model checker is based on systematically exploring the state space of a system using depth-first search (DFS) algorithm, where branching corresponds to non-deterministic choices due to parallelism or user input. Initially, Spin builds automata corresponding to each of the input modules that are part of the distributed system. The main attribute of the approach is that the model checker does not have to pre-compute the automaton for the complete system. The search is performed on-the-fly and the new states are computed as the paths leading to them are being explored. The on-the-fly approach gives Spin its scalability.

In order to check if a system satisfies a safety property ϕ, Spin checks that there are no reachable states satisfying $\neg\phi$, such states would violate the property. Model checking of liveness properties is more complex. In order to check if a system satisfies a liveness property, one has to find an infinite run (ω-run) that contains infinitely many accepting states. In finite state systems, ω-run corresponds to a lasso – a run that leads into a cycle. If the cycle contains at least one of the accepting states, the run is ω-accepting. Spin detects the ω-accepting runs using nested DFS algorithm [37], which, first, searches for an accepting state and after such state is found, performs another DFS to find out if this accepting state is reachable from itself.

The DFS algorithm utilizes the following two data structures: the stack, which corresponds to the system path that is being explored, and the visited state space, which contains all the states visited so far. = The stack is used for DFS backtracking. Note that the number of states stored on the stack is usually much smaller then the number of the visited states. Furthermore, the main operation performed by the search is determining if the state has been visited before or not. The performance of the model checker greatly depends on how fast this can be done. Not surprisingly, the visited states are stored in a hash table using a technique called bitstate hashing [33].

Abstraction is a well known way to combat the state space explosion problem. The abstraction in Spin [36] exploits the separation of the states needed for backtracking (the stack) and the visited states (the hash table). It is essential that we store the full (or concrete) state on the stack. The full state is required for computation of the alternative paths outgoing from that state upon backtracking. However, an abstract state can be stored in the hash table. When abstraction is applied, the search is not going to explore a state that is equivalent, under the abstraction, to a previously visited state. In Spin, the objects that are stored on the stack are called *tracked* and the object descriptors stored in the visited set are called *matched*.

Spin also uses internal static partial order reduction technique [31], which reduces the state space while preserving the soundness and completeness of the model checker. The technique statically determines if some of the concurrently executed transitions are commutative – result in the same state when executed in different orders. When the commutative transitions are found, one can prune the DFS tree by exploring only one execution order. Recently, due to the interest in multi-core computing, the algorithms for multi-core model checking have been added to Spin [38].

Like most classical model checkers, Spin targets verification of a system's model rather then the implementation. In particular, the model should be described using the PROMELA language. An advantage of checking the algorithmic descriptions is that they usually represent good abstractions of the system and, thus, are tractable. However, there are disadvantages. As we pointed out in Chapter 1, the correctness of the executable is what really matters. In addition, the classical model checking approach is labor intensive. First, a manual model of the complete system is built. Next, whenever there is a change in the implementation, it has to be reflected in the model.

3.2.2 Model Driven Verification

Our work extends previous work on *model-driven verification*, in which model checking was applied to the verification of *sequential* C programs [36],[26]. Model-driven verification is a form of software model checking for C programs that works by executing C code embedded in a PROMELA model. The Spin model checker [8] translates a PROMELA model (along with an LTL property to be checked) into a C program `pan.c`. This program is a model checker that checks the property in question. In a sense, Spin is really a model checker generator. Because Spin compiles models into C code, recent versions allow fragments of C programs to be embedded within PROMELA models. Each such fragment is treated as a deterministic atomic transition. This allows Spin to be used to check C programs against LTL specifications using most of Spin's search options, including bitstate hashing, and multi-core execution. Two key options not supported for embedded code by Spin are breadth-first search, which would require too much additional overhead, and partial-order reduction, which is difficult for C programs because computing a nontrivial independency relation is hard.

A significant limitation of model-driven verification is that each fragment of embedded C code is executed as an atomic transition by Spin. This in turn means that it is hard to

- check properties such as assertions and invariants at control points *within* the embedded C code
- interrupt the control flow within a C function to simulate, say, a device interrupt or an asynchronous reset
- explore interleavings of more than one fragment of C code, which is needed in order to check multi-threaded C programs.

There exist ad hoc solutions to the first two issues [26]. However, they do not solve the core problem.

3.3 Model Checking C programs with pancam

Our approach to checking a concurrent C program with Spin is to first translate the program into bytecode for the Low Level Virtual Machine (LLVM) compiler infrastructure [42]. This bytecode is then checked by executing it within the context of an explicit-state model checker by using a virtual machine interpreter. In a sense, this approach is similar to Java Pathfinder (JPF) [65] - an explicit state model checker for Java bytecode. However, unlike JPF, we do not integrate the model checker with the bytecode interpreter. Instead, our virtual machine executes bytecode as directed by Spin.

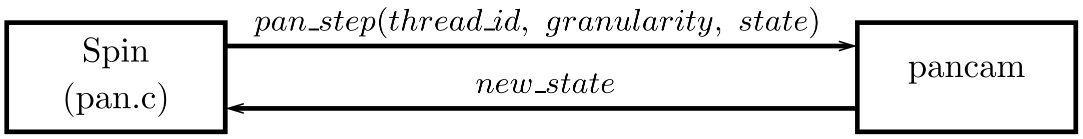


Figure 3.1: Interaction of Spin and pancam

Fig. 3.1 describes the high level idea behind the pancam framework (The actual implementation is described in the next two sections). The interpreter provides `pan_step(state, thread_id, granularity)` method that is called by Spin to execute the next transition of thread `thread_id` from the given state. The granularity of the transition can be customized. pancam can execute a single bytecode instruction, a basic block, run until a global variable is accessed, or rely on internal partial order reduction technique to determine the stopping point. In a sense, therefore, Spin *orchestrates* the search by deciding which thread to execute next, by storing visited states in its hash table, and by restoring a previous state during a backtracking step. This division of labor allows us to freely benefit from Spin's unique abilities, notably its scalability, search heuristics, and, lately, the capability to deploy it on multi-core CPUs [35]. The C language does not have any built-in primitives for concurrency, so our framework provides support for the constructs from the standard `pthreads` library such as mutexes and condition variables. Even though the dynamic thread creation is not yet fully implemented in pancam, the extension can be organically incorporated into the framework. The only limitation would be on the total number of threads, which should not exceed 255 (the bound imposed by Spin).

```
#include <pthread.h>
struct pa_desc {
  volatile int *f0, *f1 ;
  int last ;
} ;
...
volatile int pa_f0, pa_f1, pa_last ;
...
void pa_desc_lock(struct
pa_desc *d) {
  for (*d->f0=1, pa_last=d->last;
       *d->f1==1 && pa_last==d->last;
      ) ;
}
...
int count = 0 ;
void threadx_critical(void) {
  count++ ;
  ... // critical section
  count-- ;
}
```

```
void * thread1_main(void *args){
  struct pa_desc d ;
  pa_desc_init(&d, 1) ;
  for (;;) {
    pa_desc_lock(&d) ;
    threadx_critical() ;
    pa_desc_unlock(&d) ;
  }
  return NULL ; /*NOT REACHED*/
}

/* pancam helpers */

void init(void) {
  pa_f0 = pa_f1 = pa_last = 0 ;
}
Bool check_exclusion(void) {
  return (count <= 1) ;
}
```

Figure 3.2: Excerpt of Peterson's Algorithm from the Wikipedia

Fig. 3.2 shows the C implementation of the Peterson's mutual exclusion algorithm that appears in the Wikipedia entry [3]. In the original implementation, the two highlighted occurrences of `volatile` were missing, causing a potential race condition. Let us illustrate how our framework works on this example. To simplify the statement of the mutual exclusion property, we have added an additional variable `count` as shown. The property to be checked is mutual exclusion, which is defined by the boolean valued function `check_exclusion`.

Our tool first compiles this program into LLVM bytecode, using the `llvm-gcc` compiler. LLVM bytecode is like typed assembly language; a sample appears in

```
define void @pa_desc_lock(%struct.pa_desc* %d) {
entry:
       %tmp1 = getelementptr %struct.pa_desc* %d, i32 0, i32 0
       %tmp2 = load i32** %tmp1
       volatile store i32 1, i32* %tmp2
       %tmp4 = getelementptr %struct.pa_desc* %d, i32 0, i32 2
       %tmp5 = volatile load i32* %tmp4
       volatile store i32 %tmp5, i32* @pa_last
       %tmp8 = getelementptr %struct.pa_desc* %d, i32 0, i32 1
       %tmp9 = load i32** %tmp8
       br label %bb6
bb6:
       %tmp10 = volatile load i32* %tmp9
       %tmp11 = icmp eq i32 %tmp10, 1
       br i1 %tmp11, label %cond_next, label %return
cond_next:
       %tmp15 = volatile load i32* %tmp4
       %tmp16 = volatile load i32* @pa_last
       %tmp17 = icmp eq i32 %tmp15, %tmp16
       br i1 %tmp17, label %bb6, label %return
return:
       ret void
}
```

Figure 3.3: LLVM bytecode for function `pa_desc_lock`

Fig. 3.3, which shows the bytecode corresponding to the `pa_desc_lock` function shown in Fig. 3.2.

To check the C code for Peterson's algorithm with our tool, we use a PROMELA model to make appropriate calls to schedule the threads via our virtual machine. Fig. 3.4 shows a Spin model for checking the program in Fig. 3.2. The `c_decl` primitive is used to declare external C types and data objects that are used in the embedded C code. For simplicity, we assume the declarations needed by our model are in the header file `pancam_peterson.h`. Next, the `c_track` declarations are *tracking* statements, which are discussed below. The PROMELA process `init` defines the initialization steps for the Spin model: as shown, they consist of initializing the interpreter (by calling `pan_setup()`), registering an invariant (defined by the C function `check_exclusion`) with the interpreter, performing one-time initialization of the C program (`pan_run_function()`), creating and starting the threads, and, finally, starting one PROMELA process for each thread. As shown, each PROMELA process then consists of repeatedly executing a single step of the associated thread (by calling `pan_step`) provided that the thread is enabled. Note that we only pass the `thread_id`) and the `granularity` explicitly.

Passing the state vector as a parameter would be inefficient. A single contiguous region of memory starting at address `csbuf` and occupying `CS_SIZE` bytes is used for storing the program state. The buffer has to be visible to both Spin and pancam. pancam reads the state vector to execute the transition and updates it to reflect the new state. During its depth first search[1], whenever Spin reaches a state with no new successors, it backtracks to the most recent state that has not been fully explored. For PROMELA variables, restoration of earlier values when backtracking is automatic, since they are stored in the state vector maintained by Spin. However, the model checker also needs explicit knowledge of the region of memory where the program state buffer is stored, so that it can copy and restore this memory during its backtracking search. This knowledge is provided through the `c_track` declaration as shown in the figure.

[1]Spin currently supports execution of embedded C code only when using depth first search mode.

```
c_decl {
\#include "pancam_peterson.h"
}
c_track "csbuf" "CS_SIZE" "Matched";
init() {
  c_code {
      pan_setup() ;
      pan_register_invariant("check_exclusion") ;
      pan_run_function("init") ;
      pan_start_thread(0, "thread0_main", NULL) ;
      pan_start_thread(1, "thread1_main", NULL) ;
  } ;
  run thread0() ;
  run thread1()
}
proctype thread0() {
  do
  :: c_expr{pan_enabled(0, garanul)} -> c_code(pan_step(0));
  od
}
proctype thread1() {
  do
  :: c_expr{pan_enabled(1, garanul)} -> c_code(pan_step(1));
  od
}
```

Figure 3.4: Spin driver for executing pancam on Peterson's Algorithm

In using our tool to verify the Wikipedia C implementation of Peterson's protocol, we discovered a bug in the implementation. The bug is interesting because it manifests itself when the code is compiled with optimization enabled. The problem arose from the fact that certain global variables were not originally marked as `volatile` (as indicated by the shaded keywords in Fig. 3.2). As a result, the optimized bytecode reused stale values read earlier. For example, in the procedure `pa_desc_lock` from Fig. 3.3, all the instructions that occur after the second store were removed, leading to scenarios where mutual exclusion was violated. We have since fixed the Wikipedia entry.

3.4 Addressing State Space Explosion

Not surprisingly, the biggest challenge in using a tool such as pancam is the problem of state space explosion. Even though our main interest is in checking small embedded C programs, the typical state vectors we encounter are much larger (of the order of hundreds or even thousands of bytes) as compared to typical PROMELA models (whose state vectors are smaller by one or two orders of magnitude). In addition, because a single line of C may translate into many steps of bytecode, a naive exploration of all interleavings of a set of threads would quickly make even the smallest of problems intractable. To address these issues, pancam uses the following three techniques.

- It allows users to provide data abstraction functions.
- pancam provides the ability for the user to enforce context-switch bounding

(see below).

- The framework employs an algorithm that performs an on-the-fly partial order reduction to decrease the number of context switches without losing soundness of checking.

We describe the first two of these techniques in the rest of this section; our reduction method is described in Section 3.5.

3.4.1 Abstraction

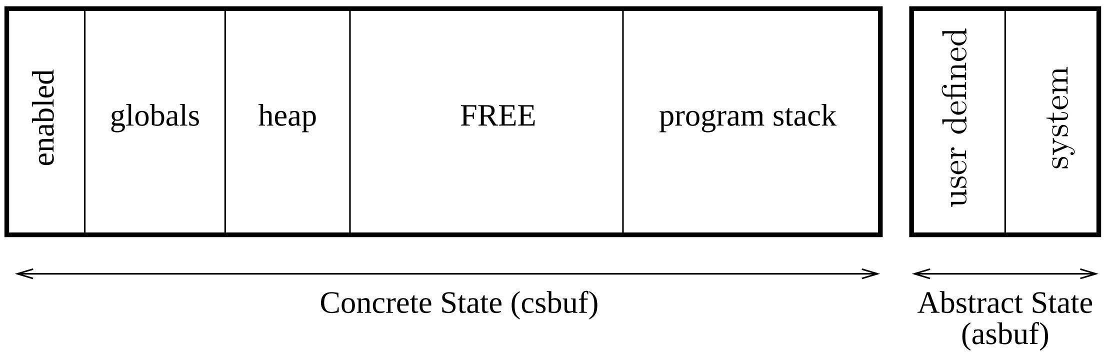


Figure 3.5: How state is maintained by pancam

Fig. 3.5 shows the layout of the program state as it is stored by pancam. Even small finite state C programs require a substantial concrete state buffer. The abstraction technique described in this section targets both the number of the states that have to be explored as well as the size of each state.

Along with updating the concrete state of the program (`csbuf`), pancam also provides an abstract state (`asbuf`). An abstract state consists of a user-defined region and additional system information such as the program counter and the fields that specify which threads are currently blocked. To compute the user-defined part, pancam interprets function `compute_abst()` that is supplied by the user of the tool and is application specific. The `compute_abst()` function updates a given memory region based on the values of the global variables of the program. Our virtual machine ensures that this function is interpreted after every Spin transition.

The ability of pancam to support abstractions is derived from the distinction between *tracked* and *matched* objects in Spin (see Section 3.2.1). As discussed, a tracked data object is stored on the stack used by Spin's depth first search (DFS), so that an earlier state of that object can be restored on each backtracking step during the DFS. In almost all cases, any data that changes during model checking should be tracked. Therefore, it is required that `csbuf` is stored on the stack. A matched object, on the other hand, is one that is a part of the state descriptor that Spin uses to determine if a state has been seen before. If bitstate hashing is used for keeping track of the visited states, the descriptor is stored in the hash table. By declaring `csbuf` to be tracked but not matched, we can therefore exclude it from the state descriptor. Then, to ensure that Spin searches the abstract state space, the abstract state `asbuf` is declared to be matched. This entails the direct space reduction since the set of visited states consists of the abstract states rather then the concrete states. Additionally, the abstraction leads to reduction in the number of explored states since the search is not going to explore computations outgoing from a state that is equivalent, under the abstraction, to a previously visited state.

```
c_decl {
  int last_proc = -1 ;
  int nswitch = 0 ;
  int MAX_SWITCH = -1 ;

  Bool pan_enabled_cb(int p) {
    int i ;

    if (!pan_enabled(p)) /* thread p is disabled */
      return FALSE ;

    if (last_proc == p) /* no context switch */
      return TRUE ;

    /* Check if bound not specified, or not reached */
    if ((MAX_SWITCH < 0) || (nswitch < MAX_SWITCH))
      return TRUE ;

    /* We have exhausted the context switch bound, and this
    **  thread is not the last one that was executed.  Allow
    **  it only if the other thread is disabled.
    */

    /* Check if any other thread is enabled */
    for (i=0 ; i<ThreadCount ; i++)
      if ((i != p) && pan_enabled(i))
        return FALSE ;
    /* all other threads are disabled, so don't preempt */
    return TRUE ;
}

c_track "&nswitch"    "sizeof(int)"  "UnMatched";
c_track "&last_proc"  "sizeof(int)"  "UnMatched" ;
```

Figure 3.6: Code for implementing context bounding with pancam

3.4.2 Context-Bounded Checking

The idea in context-bounded model checking [56, 48, 50] is to avoid state space explosion in multi-threaded programs by enforcing an upper bound on the number of allowed preemptive context switches. A context switch from process p to process q is called preemptive if process p is enabled (and could therefore continue execution if the context switch did not occur). Experience with context-bounded model checking suggests that, in most cases, errors in multi-threaded software typically have shortest counterexample traces that require only a small number of context switches [48]. Thus exhaustive exploration of runs with a small budget of allowed context switches has a good chance of finding errors.

To extend our tool with support for context-bounded search, we change the top-level Spin model that orchestrates the run by replacing calls to `pan_enabled(p)` (which check if thread `p` is enabled) by calls to the function `pan_enabled_cb(p)` (which additionally checks the condition for context-bounding). Fig. 3.6 shows the C code for the function `pan_enabled_cb`. As shown, we add two additional integers `last_proc` and `nswitch` to the state space (but note that these variables are only tracked, and not matched). It is not hard to show that by using it to replace the original `pan_enabled` function, (and by appropriately updating `last_proc` and `nswitch` whenever a thread is executed) we achieve the desired effect of limiting the number of preemptive context switches to the user-provided bound of `MAX_SWITCH`.

3.5 On-the-fly Dynamic Partial Order Reduction

As described in Section 3.3, our tool uses a SPIN model to orchestrate the state space search by choosing, at each step, a thread to execute, and executing its next transition by invoking the virtual machine. An exhaustive search along these lines would require exploring all possible interleavings of the threads in the program, which is intractable for all but the smallest of programs. A common technique used to deal with the problem is partial order reduction [52, 15]. Intuitively, partial order reduction works by reducing the number of context switches, exploiting the fact that transitions in different threads are often independent (in the sense that the order in which they occur does not affect visible program behavior).

Most partial order methods described in the literature are *static* in the sense that they determine independence of transitions by analyzing program text. Such analyses are, however, not effective with C programs, and typically allow only very simple and conservative independence relations to be computed. For C programs, therefore, it is more instructive to look at *dynamic* partial order reduction methods[23],[27], in which independence relationships are computed dynamically, during a model checking run. For example, one of the simplest approaches to dynamic partial order reduction is to only allow a context switch after an update or an access to a global memory location.

In the context of our tool, however, there is one additional complication caused by the fact that the model checking engine (SPIN) treats the model as having a single transition (denoted by the function `pan_step`). In particular, this means that support for partial order reduction therefore requires either exposing additional pancam state (which would require modification of SPIN), or for the reduction to be implemented entirely within pancam. We adopt the latter strategy. pancam performs partial order reduction on the state space by allowing a thread i to execute a sequence of more than one instruction as part of a single SPIN transition from a state s. We refer to such a sequence of instructions as a “superstep”. Since the model checker only sees the first and last states of a superstep, the intermediate states are hidden from the model checker, which in turn reduces the number of interleavings to be explored (and therefore the number of states and transitions).

Of course, as with traditional partial order reduction, there are certain conditions that must be satisfied by such supersteps in order to preserve soundness of model checking. Section 3.5.1 and Section 3.5.2 present the formal foundations of the superstep reduction and describe a set of conditions under which we can preserve the soundness of next-time free LTL properties. In Section 3.5.4 and Section 3.5.3, we show how it can be applied to checking of multithreaded programs and to pancam in particular.

3.5.1 Preliminaries

A *state transition system* M is a tuple $\langle\ S,\ T,\ S_0,\ L,\ AP\ \rangle$. Where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; T is a set of transitions such that for each $\alpha \in T,\ \alpha \subseteq S \times S$; $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

A transition $\alpha \in T$ is *enabled* in a state s if there is a state s' such that $\alpha(s,\ s')$ holds. We assume that transitions are deterministic and use the notation $s' = \alpha(s)$ instead of $\alpha(s, s')$. We will use *enabled*(s) to denote the set of transitions enabled at s.

A *path* from a state s_0 in a state transition system is a finite or infinite sequence of states and their labeled transitions $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1}$ ... such that it satisfies the following requirement: for every i, $\alpha_i(s_i, s_{i+1})$ holds. Note that a prefix of a path is also a path. We use $\mathcal{E}(M,\ s)$ to denote the set of all maximal paths of M starting at s. Given a finite path λ, we use $last(\lambda)$ to denote the last state of σ.

A *string* is a sequence of transitions from T. Let T^* be the set of all the finite and infinite strings over T. Let Λ be a string and i be a natural number. We use $\Lambda[i]$ to refer to the i^{th} transition of the string. If Λ is finite, we use $|\Lambda|$ to denote its size. Since we assume that the transitions are deterministic, a string and a state pair represent a path.

Let s be a state and Λ be a finite string. Define $\Lambda(s) = \perp$ if $\neg en_M(s, \Lambda)$, where $\perp \notin S$. Otherwise, we use $s' = \Lambda(s)$ to denote that s' is the last state on the finite path that can be obtained by applying the transitions of Λ to s. We use $\Pi(s,\ \Lambda)$ to denote such path. For a finite path λ, we use $tr(\lambda)$ to denote a string Λ such that $\lambda = \Pi(s,\ \Lambda)$.

Definition 7. *For a transition system $M = \langle\ S,\ T,\ S_0,\ L,\ AP\ \rangle$ and a non-empty string $\Lambda \in T^*$, Λ is* **enabled** *in M at $s \in S$, denoted by $en_M(s, \Lambda)$, if and only if there exists a path σ in M starting at s: $tr(\sigma) = \Lambda$.*

Definition 8. *Let Λ be a finite string and s be a system state, then we define a* **superstep transition**, *denoted by $s \rhd \Lambda$, to be a transition $\{(s, s')\} \subseteq S \times S$ such that $s' = \Lambda(s)$. In other words, $s \rhd \Lambda$ summarizes the execution of the path $\Pi(s,\ \Lambda)$. Note that when $|\Lambda| = 1$, $s \rhd \Lambda = \Lambda[1]$.*

Let X and Y be two finite strings then $X \setminus Y$ denotes a string obtained from X by removing from it all the transitions that occur in Y, which is defined recursively. Let ϵ denote the empty string; then $X \setminus \epsilon\ \equiv\ X$. For a transition a, $(X \setminus a = V \circ W)\ \equiv$ $((X = V \circ a \circ W\ \wedge\ a \notin V)\ \vee\ (X = V \circ W\ \wedge\ a \notin X))$ and $X \setminus (a \circ W)\ \equiv\ (X \setminus a) \setminus W$.

An *independence relation* $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and each $(\alpha, \beta) \in I$ such that $\alpha, \beta \in enabled(s)$.

1. $\alpha \in enabled(\beta(s))$ (enabledness condition).

2. $\alpha(\beta(s)) = \beta(\alpha(s))$ (commutativity condition).

The *dependency relation* D is the complement of I, namely $D = (T \times T) \setminus I$.

Given an atomic proposition p in AP and a state s in S, we use $p(s)$ to denote the value of p in the state s. A transition α is *visible* at a state s with respect to an atomic proposition p, if for a state $s' : s' = \alpha(s)$, $p(s) \neq p(s')$ - a transition is visible when its execution changes the value of the proposition. A transition is *invisible* if it is not visible. Denote by $vis(\lambda)$, where λ is either finite or infinite path, the projection of λ onto the transitions visible with respect to any of the propositions in AP, at the corresponding states of the path.

The concept of stuttering refers to a sequence of identically labeled states along a path. Two infinite (finite) paths $\sigma = s_0 \overset{\alpha_0}{\rightarrow} s_1 \overset{\alpha_1}{\rightarrow} ...$ and $\rho = r_0 \overset{\beta_0}{\rightarrow} r_1 \overset{\beta_1}{\rightarrow} ...$ are *stuttering equivalent*, denoted $\sigma \sim_{st} \rho$, if there are two infinite (finite) sequences of positive integers $0 = i_0 < i_1 < i_2 < ...$ and $0 = j_0 < j_1 < j_2 < ...$ such that for every $k \geq 0$,

$L(s_{i_k}) = L(s_{i_k+1}) = ... = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = L(r_{j_{k+1}-1})$.

Intuitively, two paths are stuttering equivalent when they can be partitioned into infinitely many blocks, such that all the states in the corresponding blocks from the two paths are identically labeled.

Let M be the system $\langle\ S,\ T,\ S_0,\ L,\ AP\ \rangle$ and let M' be the system $\langle\ S',\ T',\ S_0,\ L,\ AP\ \rangle$. Then we say that M and M' are *stuttering equivalent* if and only if

- M and M' have the same set of initial states.
- For each path σ of M that starts from an initial state s in S_0, there exists a path σ' of M' starting from the same initial state s such that $\sigma \sim_{st} \sigma'$.
- For each path σ' of M' that starts from an initial state s in S_0, there exists a path σ of M starting from the same initial state s such that $\sigma' \sim_{st} \sigma$.

Let M and M' be two stuttering equivalent systems. It can be shown [15] that for any LTL formula $\mathbf{A}f$ without the next time operator, and every initial state $s \in S_0$, $s \models \mathbf{A}f$ in M if and only if $s \models \mathbf{A}f$ in M'.

3.5.2 Superstep Partial Order Reduction

A partial order reduction (POR) algorithm constructs a reduced state graph, referred to as M', stuttering equivalent to the given graph M. The SPOR is similar to the classical POR presented in [15]. In the classical approach, one starts with the input graph M and iteratively applies a reduction procedure. At each step, a state s of the graph is considered and a subset of transitions enabled at s is selected; this subset is referred to as the ample set. The rest of the transitions outgoing from s can be removed from the graph without sacrificing the soundness and completeness of the LTL model checking. Unlike the classical POR, which considerers a subset of enabled transitions at a state s, SPOR selects a subset of finite paths enabled at s;

such subset is called a *cap* set. Further, each path from a cap set is collapsed into a single *superstep* transition.

Like the classical POR, SPOR can be applied on-the-fly, when the construction of the reduced system and model checking are combined. This approach is more efficient when applied to implicit state model checkers since the transitions and states of the input transition graph are constructed on demand. On-the-fly model checking makes it possible to identify a violation before completing the construction of the state graph. In addition, SPOR does not assume any particular order in which the states are visited. However, if the goal is to apply the reduction to the complete graph, it is more efficient if no node is visited before its dominator (assuming only one initial state), which ensures that we never explore the states that will get removed in the future. In particular, the reduction can be combined with both BFS and DFS model checking algorithms; however as we will show later, the overhead of using DFS is greater.

Given a state s of transition graph M, let $Cap(s)$ be a subset of finite strings enabled at s. The reduction procedure on state s removes all the transitions outgoing from s. Next, it adds the following new transitions to the graph: $\{\ s \rhd \Lambda\ :\ \Lambda \in Cap(s)\ \}$. The reduction does not increase the number of states; indeed, it potentially decreases the number of reachable states. Next, we present the set of conditions that the set $Cap(s)$ should satisfy in order to ensure that stuttering equivalence is preserved.

Let $X \ll Y$ denote that X is a (noncontiguous) subsequence of Y; formally $X \ll Y \ \equiv\ (\exists Z : Y \setminus Z = X)$.

Definition 9. *Given finite non-empty strings Θ and Λ such that $\Lambda \ll \Theta$, Λ is a* **core** *of Θ at a state s if and only if*

1. $\forall\ t, t' :\ t \in (\Theta \setminus \Lambda),\ t' \in \Lambda\ :\ I(t, t')$

2. $vis(\Pi(s,\ \Lambda)) = vis(\Pi(s,\ \Theta))$

3. $|vis(\Pi(s,\ \Lambda))| \leq 1$

Note that Θ is a core of Θ at any state $s \in S$.

Lemma 3. *If Λ is a core of Θ at s, then*

1. $\Theta(s)\ = (\Lambda \circ (\Theta \setminus \Lambda))(s)$

2. $\Pi(s,\ \Theta)\ \sim_{st}\ \Pi(s,\ \Lambda)\ \sim_{st}\ \Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$

Please, refer to B for the proof of the lemma.

Definition 10. *Let s be a state of a transition system M and $Cap(s)$ be a subset of strings enabled at s. Then $Cap(s)$* **covers** *s if and only if $\forall \sigma \in \mathcal{E}(M,\ s)$:*

$$\begin{aligned} \exists\ \Theta,\ \Lambda\ :\ \ & \Theta \textit{ is a finite non-empty prefix of } tr(\sigma) \\ & \wedge\ \Lambda \in Cap(s) \\ & \wedge\ \Lambda \textit{ is a core of } \Theta \textit{ at } s. \end{aligned}$$

Intuitively, $Cap(s)$ covers s if for every non-empty outgoing prefix of transitions Θ, there exists another prefix of transitions Λ such that exploring Λ would cover the sequence of states undistinguishable from the one that would be seen if one is to take Θ. Thus, Θ can be pruned away. First, let's show that all the states that get explored after following the path $\Pi(s,\ \Theta)$ will also get explored by following $\Pi(s,\ \Lambda)$.

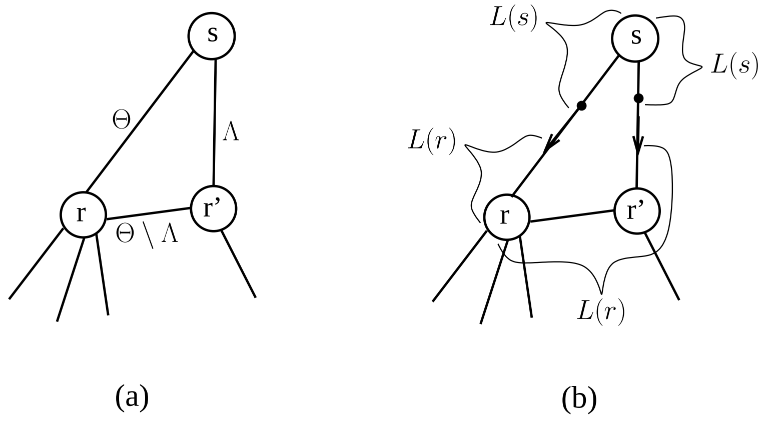


Figure 3.7: Correctness of SPOR

According to Lemma 3.1, since Λ and Θ only differ in independent transitions, $\Lambda \circ \Theta \setminus \Lambda$ is enabled at s and will result in the same state as Θ. Refer to Fig. 3.7(a) for the illustration, where the common state is labeled by r. Second, we need to ensure that the sequences of states produced by Θ and $\Lambda \circ \Theta \setminus \Lambda$ cannot be distinguished from one another with respect to any LTL property, formally stated by Lemma 3.2: $\Pi(s,\ \Theta)\ \sim_{st}\ \Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$. By definition of core, both $\Pi(s,\ \Lambda)$ and $\Pi(s,\ \Theta)$ share the same visible transition; and at most one such transition is allowed. Thus, as depicted on Fig. 3.7(b), all states on the paths $\Pi(s,\ \Theta)$ and $\Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$ can be partitioned into two sets Seq and Req:

- $\forall s' \in Seq : L(s') = L(s)$
- $\forall r' \in Req : L(r') = L(r)$

Since all three strings: Θ, Λ, and $\Theta \setminus \Lambda$, are finite, the paths are stuttering equivalent. Note that even after the path $\Pi(s,\ \Lambda)$ is substituted by a single superstep transition $s \triangleright \Lambda$, the stuttering equivalence is preserved.

Definition 11. *Given system $M = \langle\ S,\ T,\ S_0,\ L,\ AP\ \rangle$ and a set of superstep transitions $\overline{Cap} = \{s \rhd \Lambda\ :\ s \in S,\ \Lambda \in Cap(s)\}$ such that $\forall s \in S : Cap(s)$ covers s. The* **reduced system** *M', obtained by application of the reduction procedure to all the states in M, is formally defined as follows: $M' = \langle\ S,\ \overline{Cap},\ S_0,\ L,\ AP\ \rangle$.*

Note that the set of all the transitions enabled at a state s satisfies $Cap(s)$. Thus, the definition of a reduced system applies to systems in which the reduction procedure is not necessarily applied to all the states; and $(\forall s \in S :\ Cap(s) = enabled(s)) \ \rightarrow\ \overline{Cap} = T$. Our main result states that our superstep reduction is sound:

Theorem 3. *The transition graphs M and M' are stuttering equivalent.*

The proof is shown in appendix B.

3.5.3 Application to Multithreaded Programs

For convenience, we consider programs with a finite number of deterministic threads (or processes), where the only source of nondeterminism comes from thread scheduling. We also assume that each transition can access at most one global memory location. This assumption is safe to make about the LLVM bytecode, which uses designated instructions *store* and *load* to access memory. We say that two transitions *conflict* if both access a common memory location and at least one of them is a write. Under the assumption that one thread may enable or disable another only by means of mutexes, which are a type of a shared object, the absence of a conflict between transitions implies independence as long as the transitions do not belong to the same thread. Note that not only the conflicting transitions may produce different results depending on the execution order, but they may also disable each other.

Definition 12. *For a given state s, $ThreadedCap(s)$ is a finite subset of strings enabled at s such that, for every thread i enabled at s, it includes exactly one string Λ_i^s such that Λ_i^s only includes the transitions of thread i and satisfies the following three requirements:*

1. **Superstep Size** *Λ_i^s must be finite and contain at least one transition. The check for finiteness can be implemented conservatively by setting an upper bound on the number of transitions in a superstep sequence or the number of loop heads within the sequence.*

2. **Independence** *Only the very last transition of the path Λ_i^s conflicts with any of the transitions in Λ_k^s for any thread $k \neq i$.*

3. **Visibility** *At most one transition which changes the value of any of the atomic propositions is allowed in Λ_i^s. If exists, it must be the very last transition of the superstep sequence.*

For example, consider Fig. 3.8, which depicts the superstep sequences of the three program threads (Th_1, Th_2, and Th_3) from the program state s. By the requirements above, $\alpha_1,\ \beta_1,\ \gamma_1,\ \gamma_2$ are independent of each other. Whereas transitions $\alpha_2,\ \beta_2,\ \gamma_3$ may be interdependent.

Lemma 4. *For any state s, the set $ThreadedCap(s)$ covers s.*

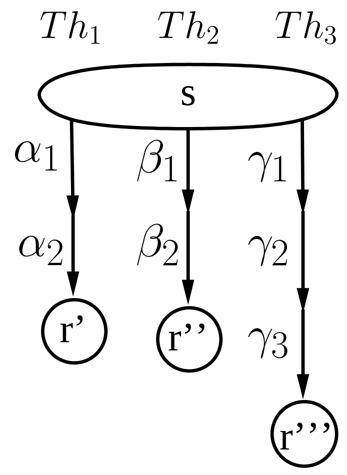


Figure 3.8: Application of SPOR to threads

Thus, whenever a model checking algorithm visits a state s, it is sufficient to only explore the successor states that result in the execution of the superstep transitions $\{s \rhd \Lambda \ : \ s \in S, \ \Lambda \in \mathit{ThreadedCap}(s)\}$. As follows directly from Lemma 4 and Theorem 3, such reduction preserves the soundness of next-time free LTL model checking. Please refer to for the proof of Lemma 4.

3.5.4 Implementation of Superstep Reduction in pancam

Next, we describe how superstep reduction is implemented as part of pancam, which piggybacks the nested depth-first search algorithm used by SPIN. One of the attractions of using SPIN's nested depth-first search is that, unlike the case with breadth-first search [27], our implementation is fully compatible with checking of liveness properties. (And, although, we do not describe it here, our method can be straightforwardly extended to cooperate with breadth-first search, if desired.)

During its state exploration, Spin issues calls to `pan_step(i)`, which, given the current state s, computes the state s' obtained by executing one or more instructions of thread i. The executed instructions form the superstep sequence Λ_i^s. The superstep size requirement guarantees that at least one instruction would be executed; consequently, unless there is a loop in the state space, $s \neq s'$. Due to the nature of depth-first search, `pan_step` will be called multiple times on the same state s. In particular, after exploring the state space in which thread i is executed from state s, Spin backtracks and attempts to execute the thread $i + 1$ from the same state s in response to which pancam computes Λ_{i+1}^s.

The pseudocode of `pan_step` is presented in Fig. 3.9. If the state s is visited by the depth-first search for the very first time, `pan_step` executes initialization routines. Further, each time Spin calls `pan_step(i)`, we compute the superstep sequence for thread i by interpreting the enabled instructions of thread i one by one. On each iteration, we check that addition of the corresponding instruction to the sequence does not violate any of the requirements stated above (in practice, the checks are only required for the instructions that access a global program location).

The most non-trivial check is the verification of the independence condition for which one could use various static and dynamic methods. Fig. 3.10 presents the dynamic independence check employed by pancam. Due to the nature of the independence requirement, the superstep of one thread depends on the transitions that constitute the supersteps of the other threads. An eager approach to this problem is to compute the supersteps for every thread the very first time the state s is visited (with the request to take step on thread one) and use the precomputed

```
ConflictType = { CONTINUE, POST_STOP, PRE_STOP }

pan_step(ThreadID i) {
  superstep_length = 0;

  if (not backtracking) {
    init_independence_tester();
  }

  while (true)  {
    $tr_i$ = get_next_instruction(i);
    ConflictType error = test_for_independence(i, $tr_i$);
    if (error == PRE_STOP) break;
    execute_instruction($tr_i$);
    superstep_length++;
    if (superstep_length $\geq$ MAX_SUPERSTEP_LENGTH) break;
    if (error == POST_STOP) break;
    if (is_proposition_modifying($tr_i$)) break;
} }
```

Figure 3.9: Pseudocode of `pan_step` with superstep reduction.

supersteps on all the subsequent visits to the same state (when Spin backtracks to take step on the other threads). However, this solution leads to inefficiencies since computing the supersteps effectively entails computation of the successors of the state s. Storing the successor states along with the current state leads to a large space overhead. Recomputing the successor states, on the other hand, would impair the running time.

The solution we present computes the supersteps lazily - whenever `pan_step(i)` is called, it only computes the superstep for thread i. To convey the information about the supersteps which have already been computed, we store additional information along with the program state on the depth-first search stack. For each state s and

```
init_independence_tester() {
  for (every enabled thread k) {
    AccessTable$^s_k$.add( get_access_pair( get_next_instruction(k) ) );
  }
}

test_for_independence(ThreadID i, Instruction tr$_i$) {
  ai = get_access_pair(tr$_i$);
  for ( all threads k :  k $\neq$ i ) {
    for (all ak $\in$ AccessTable$^s_k$) {
      if (conflict(ai, ak)) {
        if (ak $\neq$ last_of(AccessTable$^s_k$)) {
          return PRE_STOP;
        } else {
          if (tr$_i$ $\neq$ first_of($\Lambda^s_i$)) AccessTable$^s_i$.add(ai);
          return POST_STOP;
  } } } }
  if (tr$_i$ $\neq$ first_of($\Lambda^s_i$)) AccessTable$^s_i$.add(ai);
  return CONTINUE;
}
```

Figure 3.10: Pseudocode of the independence condition tester

each thread i, we store `AccessTable`^s_i - the list of location and access type pairs. Each instruction of Λ_i^s that accesses a global is represented by a pair $(\mathtt{l}, \mathtt{ty})$; it records the global location `l` that is accessed and the flag `ty` stating whether the transition is a read or a write. `AccessTable` is not stored as the part of the state tracked by Spin but maintained externally within pancam VM since the data it stores is updated each time the state is visited.

The very first time state s is visited, `init_independence_tester`() initializes the `AccessTable`^s_i of each enabled thread i with the access information derived from the very first instruction to be executed on the thread i. Further, before adding an instruction $\mathtt{tr_i}$ to the superstep sequence of thread i, pan_step consults with $\mathtt{test\_for\_independence}(\mathtt{i}, \mathtt{tr_i})$ to ensure that the independence condition is met. test_for_independence may return three different values. CONTINUE means that the instruction can be added to the superstep Λ_i^s since it does not conflict with any instructions in Λ_k^s for all threads $k \neq i$. POST_STOP means that $\mathtt{tr_i}$ introduces a conflict with some other thread, but adding it to Λ_i^s does not violate the independence requirement as long as it is the very last transition of Λ_i^s. Finally, PRE_STOP means that adding $\mathtt{tr_i}$ to Λ_i^s leads to a violation since the transition with which it conflicts is not the very last transition of thread k for some $k \neq i$; thus, $\mathtt{tr_i}$ must not be executed. Due to the initialization of $\mathtt{AccessTable_i^s}$, it is not possible to have a PRE_STOP on the very first transition of any of the threads; thus, the superstep size requirement is met - pan_step always executes at least one transition. Finally, $\mathtt{test\_for\_independence}(\mathtt{i}, \mathtt{tr_i})$ updates the $\mathtt{AccessTable_s^i}$ with the access pair derived from $\mathtt{tr_i}$ if the instruction is to be added to Λ_i^s and if it is not the very first instruction of Λ_i^s. Recall that the AccessTable is updated with the access pairs corresponding to the very first instructions of each thread as part of the initialization routine.

The above technique requires no space overhead when used as part of breadth-first search state exploration. However, when used with depth-first search, the AccessTable must be stored on the search stack. In cases when the sets are quite large, one could use approximations. For instance, one idea is to use a *coloring* abstraction, in which the memory is partitioned into regions with distinct colors, and each transition is associated with the set of colors it reads and writes.

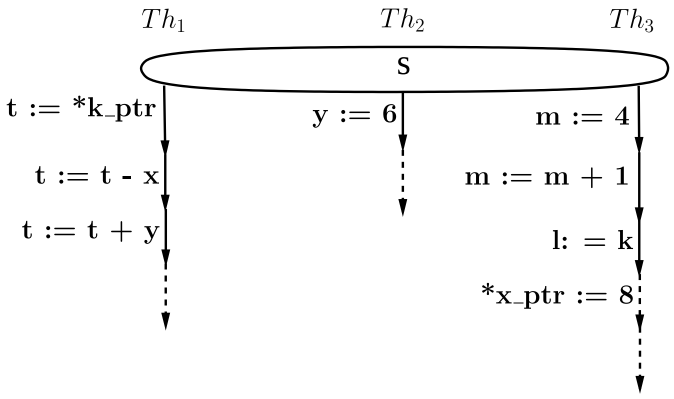


Figure 3.11: The example of the Superstep POR algorithm

Example 3. *Let us demonstrate the algorithm on an artificial example from Fig. 3.11 that depicts the instructions that the three threads can execute from the state s. The solid arrows represent the instructions that form the supersteps from the state s. We assume that the variables k, x, y, and m are global variables; t, l, x_ptr, k_ptr are local; x_ptr and k_ptr are the pointers to x and k, respectively.*

When the state s is visited for the very first time, `init_independence_tester` *initializes the* `AccessTable` *with the information derived from the very first instructions of each thread as following:*

$\texttt{AccessTable}^{s}_{1} = \ (\ (k\_ptr, read)\ )$

$\texttt{AccessTable}^{s}_{2} = \ (\ (ad(y), write)\ )$

$\texttt{AccessTable}^{s}_{3} = \ (\ (ad(m), write)\ )$

Here $ad(x)$ stands for the address in memory where the variable x is stored (`AccessTable` *stores the actual addresses of the accessed variables). After the initialization,* `pan_step` *issues calls to* `test_for_independence` *passing the instructions of Th_1 one by one. The function returns* `CONTINUE` *when passed* **t := *k_ptr** *and* **t := t - x**. *However, since* **t := t + y** *conflicts with the very first instruction of* Th_2*,* `POST_STOP` *is returned as the result of the third call. The table is updated accordingly:*

$$\texttt{AccessTable}^{\texttt{s}}_1 = ((k\_ptr, read);\ (ad(x), read);\ (ad(y), read))$$

When the depth-first search backtracks to schedule Th_2*,* `pan_step` *calls* `test_for_independence` *with* **y := 6** *as the argument. Due to the conflict with the last instruction of* Th_1*, the function returns* `POST_STOP`*, making* **y := 6** *to be the only instruction forming* Λ^s_2*. The* $\texttt{AccessTable}^{\texttt{s}}_2$ *does not need to be updated.*

Finally, when `pan_step(3)` *is called, the check for independence on the first three instructions of* Th_3 *returns* `CONTINUE`*. Note that even though both the third instruction of* Th_3 *and the first instruction of* Th_1 *read from the same memory location: it can be determined at run time that* k_ptr *equals* $ad(k)$*, no conflict is reported. However, the fourth instruction,* ***x_ptr := 8***, conflicts with the second entry in* $\texttt{AccessTable}^{\texttt{s}}_1$ *raising the* `PRE_STOP` *return code. Since the conflicting transition is not the last transition of* Λ^s_1*,* ***x_ptr := 8** *should not be included in* Λ^s_3*.*

3.6 Experimental Results

We have gathered some initial experimental results with our prototype on a few small multi-threaded C programs. Fig. 3.12 and Fig. 3.13 show results from checking two versions of the implementation of Peterson's algorithm in C, described in Section 3.3. Fig. 3.12 shows the number of states explored against varying context bounds for the version of the program with the missing `volatile` keyword bug, while Fig. 3.13 shows similar results for the version of the program without the bug. The graphs

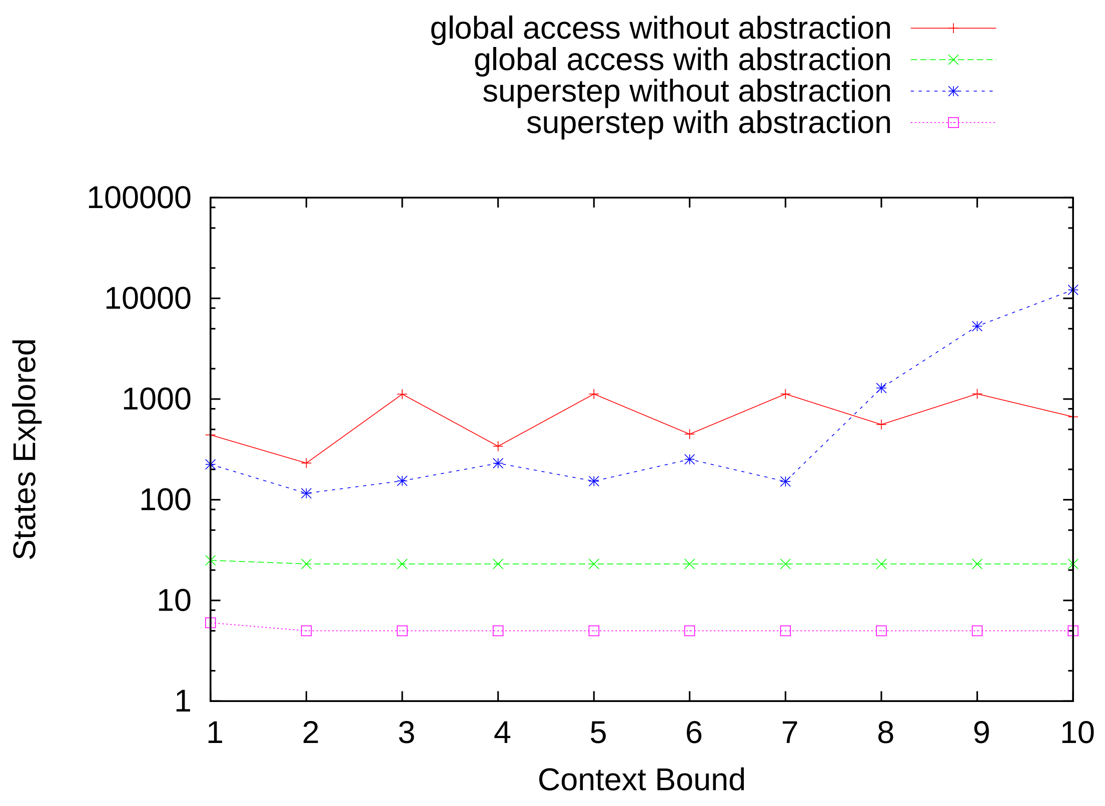


Figure 3.12: Context bounding for peterson.c with bug

also compare a heuristic that runs a thread until it makes an access to any global state (labeled "global access" in the figure) versus our superstep reduction method (labeled "superstep"). As the graphs indicate, the bug is found fairly easily in all versions, though increasing the context bound beyond a certain point makes it harder to find the bug. (This is likely a consequence of the fact that Spin uses depth-first search.) The graphs also show the benefit of an abstraction function we used which tracks only the algorithmic state of the protocol (the value of the abstract "flag" and "turn" variables).

Fig. 3.14 shows results from the "robot" benchmark example [27]. This example

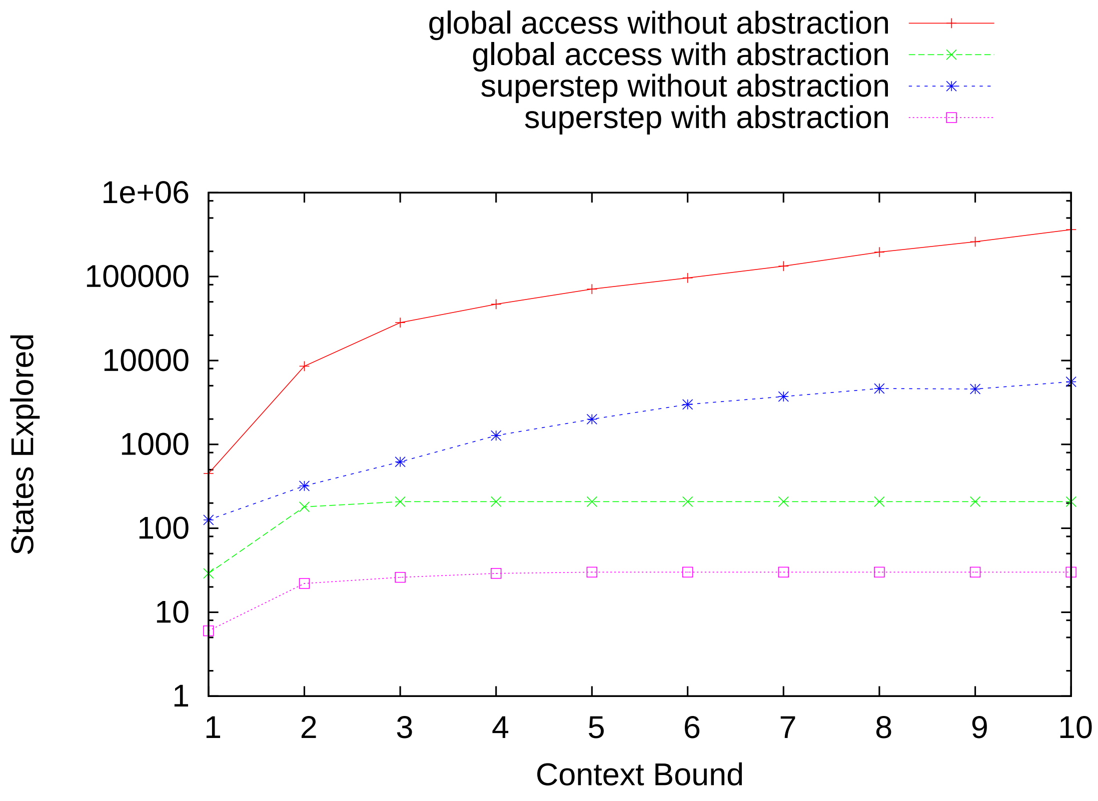


Figure 3.13: Context bounding for peterson.c without bug

consists of two threads that move across a shared board of size $N \times N$ in slightly different patterns; the program checks that the robots meet only in expected locations. As the graph shows, our superstep method provides a noticeable reduction in the number of states over the global access method, as the size of the board grows.

Table 3.1 compares the improvement of superstep reduction with respect to the global access heuristic on two more examples. The first is a C implementation of the classic dining philosophers algorithm, with varying number of philosophers (denoted by parameter n). The second example is an inter-process communication module for an upcoming NASA Jet Propulsion Laboratory mission. The module consists

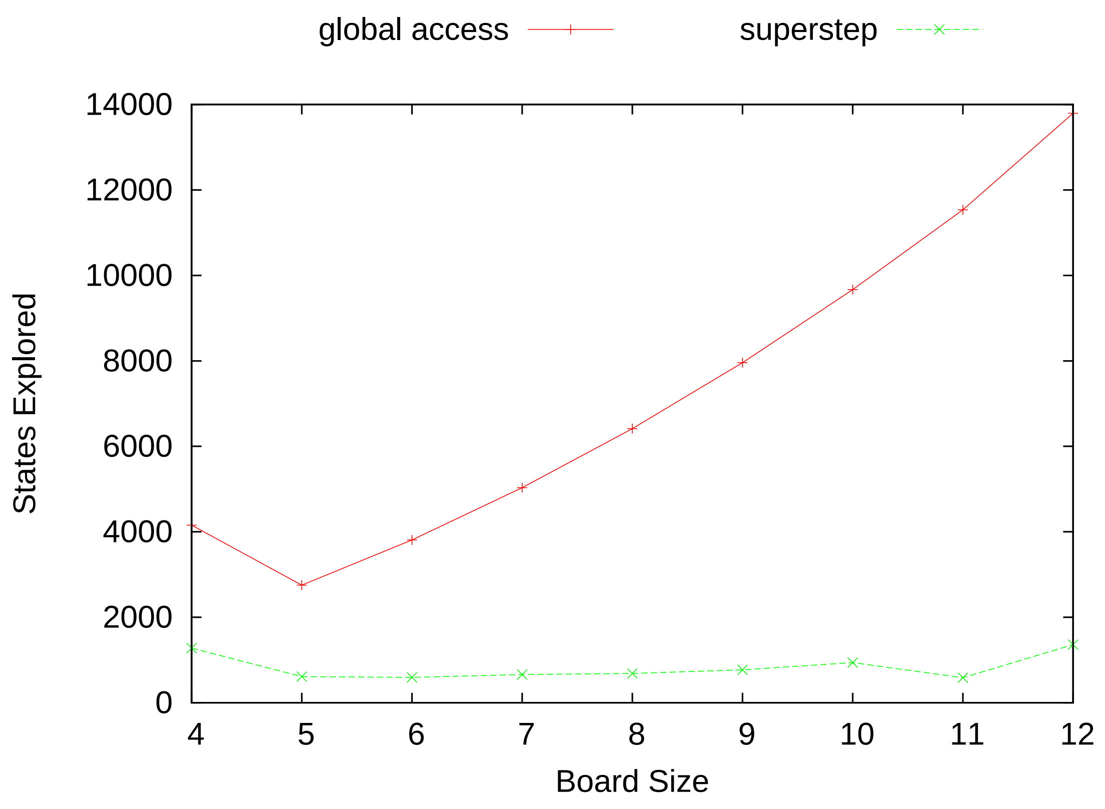


Figure 3.14: The Robot example

Benchmark	#states global access	#states superstep
Phil n=2	59	37
Phil n=3	534	380
Phil n=4	4762	3130
Phil n=5	42386	25021
IPC m=1	156863	234
IPC m=2	625359	316
IPC m=3	1529342	479
IPC m=4	!	654
IPC m=15	!	22629

Table 3.1: Other experiments with pancam

of around 2800 lines of (non-commented) C source code (including some support modules that it relies on). It implements a communication system that supports prioritized messages and provides thread-safe primitives for sending and receiving messages. To give meaningful results, we restricted the model to a single producer-consumer pair, and forced a bound of 4 context switches, while varying queue depth (denoted by m). Even with the small configuration parameters, the default global access heuristic exhausts memory resources for $m = 4$ (as denoted by the symbol !) on a machine with 32 GBytes of RAM, whereas the superstep method can handle much larger configurations (well over $m = 15$).

3.7 Related Work

There has been considerable interest in applying model checking directly to implementation code. The Bandera checker [16] translates Java programs to the input language for a model checker, while Java Pathfinder (JPF) [65] uses an approach more similar to ours, in that it interprets bytecode. However, JPF tightly integrates model checking with the virtual machine. In contrast, our tool uses the Spin model checker to orchestrate the search, using our virtual machine to execute the transitions. This allows us to inherit (for free) the various optimizations and features of Spin (both those that exist, and those yet to be invented). In spite of this loose integration, our approach is flexible; for instance, as shown in Section 3.3, adding support for bounding context-switches was done fairly easily in our tool. Using Modex [32] - a tool which extracts PROMELA models from C implementations provides similar benefits. However, the model extractor is guided by user-defined abstractions, construction of which requires a considerable manual effort.

For verification of multi-threaded C programs, the CMC tool [49] uses explicit-state model checking. One limitation of CMC, however, is that it requires a manual step by the user to convert an existing C program into a form that can be used by CMC. In contrast, by working directly on bytecode, our tool design is simpler (interpreting typed LLVM bytecode is much easier than interpreting C). In addition, we are able to detect errors introduced during compiler optimization (like the Wikipedia error in Peterson's algorithm, described in Section 3.3).

In this respect, our work is related to "WYSINWYX: What You See Is Not What You eXecute" [11], in which program analysis is applied to a model constructed from an executable. WYSINWYX framework is not coupled with any particular compiler. Another advantage of the framework is that the errors introduced by a compiler back-end would also be discovered. However, the constructed model is not precise since it has to recover information about variables and types, which is especially difficult for aggregates (i.e., structures and arrays).

Another tool for verifying C programs is VeriSoft [25], which uses *stateless* model checking. VeriSoft uses static partial order reduction, which typically results in little reduction when applied to C programs, since the independence relation is hard to compute. More recent work on dynamic partial-order reduction [23] addresses this problem. Inspect tool [66] applies the idea to verification of C programs. Since the technique is only applicable to the stateless search in which the search depth is bounded, it poses a challenge for programs whose state graphs have cycles.

More directly related to the superstep reduction presented in Section 3.5 is the work on "cartesian partial order reduction" [27], which is a method that dynamically computes independence relationships, and tries to avoid context switching whenever possible. The ideas behind cartesian partial order reduction and superstep reduction are closely related, though there are significant implementation differences. In particular, our reduction is done in the context of Spin's depth-first search. While this complicates the design somewhat, and incurs some additional memory overhead, it can be applied even when checking liveness properties. (In contrast, the cartesian reduction method was applied only in the context of checking assertion violations and deadlocks.)

Our approach to enforcing context-bounding is directly inspired by the work on the CHESS model checker for concurrent C code [50, 48]. One point of departure is that, even with fair scheduling, the CHESS model checker only checks livelocks; in contrast, our approach is able to handle general liveness properties.

Chapter 4

Conclusion

We have presented two approaches to ensuring the correctness of compiled code – compiler verification (TVI, CoVaC) and model checking of optimized bytecode (pancam).

Each of these approaches has its advantages. Model checking wins when one has to check Linear Temporal Logic properties over relatively small programs (under 10K lines of code). If pancam is chosen, we have a very high assurance that what we check is what will be executed. Of cause, since we check bytecode and not the machine code, there is still a possibility of error at the very last step (in the translation from bytecode to machine code). However, model checking of bytecode gets us very close to the ultimate goal. More importantly, pancam gives the most advantage when one has to check multi-threaded applications. Note that the presented compiler verification techniques are targeted at verification of sequential programs. Take for example the problem from Section 3.3 where a missing volatile modifier lead to a race condition. This bug would not be detected by CoVaC. Even though we can ensure that there are no miscompilations involving the volatile variables by representing the access and stores to the volatiles using the read and write instructions, there is no way of ensuring that the modifier is used in the first place. Furthermore, one may rightly argue that catching missing volatile modifiers is not in the realm of a compiler verifier. On the other hand, this error would also go undetected by the tools for source code analysis.

There are several settings in which one would go back to translation validation. One such setting is when the application being checked is large. In particular, since CoVaC performs intraprocedural analysis, it is expected to scale well when applied to modular code. Another setting is when we are only interested in verifying if the compiler optimizations preserve the semantics of the program. This is true in compiler testing, where translation validation can be used instead of unit tests. Additionally, this is the case when one prefers using a particular verification tool to check that the program satisfies the desired property. For example, suppose we have an application that has to be resilient to security vulnerabilities. Here, it might be advisable to use some existing tools targeted at checking security properties. These tools may come equipped with an extensive catalog of security properties, which may not be available to the user. This makes using the off-the-shelf verifier more efficient and effective then constructing our own list of security properties and checking for conformance with pancam.

Here is another way of contrasting the two approaches. pancam is a checker that ensures that the bytecode satisfies an LTL specification. Whereas CoVaC ensures that the bytecode obtained after the compilation is equivalent to the non-optimized version. The non-optimized source code serves as the specification for CoVaC. So, effectively, it checks conformance to a much stronger specification then that of pancam. Intuitively, in order to be uncovered by pancam, a compiler bug has to directly effect the LTL property that is being checked. Of cause, since the presented translation validation algorithms target only structure preserving optimizations, they verify a relatively small segment on the road from the model of the system to the executable. To conclude, the two techniques are orthogonal and it would not be redundant to apply both to verification of a single program.

Appendix A

CoVaC Proofs

Lemma 5. $\forall \sigma \in \mathit{Cmp}(f) \, : (\exists \sigma^S \in \mathit{Cmp}(f^S) \, : \, \sigma^S \sim_{st} \sigma{\uparrow}_S) \wedge (\exists \, \sigma^T \in \mathit{Cmp}(f^T) \, : \sigma^T \sim_{st} \sigma{\uparrow}_T).$ The claim follows directly from Rule 1 and Rule 2.

Theorem 1 Target function f^T is *a correct translation* of source function f^S if and only if there exists a *witness* comparison graph $f = f^S \boxtimes f^T$. In addition, if f^T is a correct translation of f^S then every comparison graph $f = f^S \boxtimes f^T$ is a witness of correct translation.

Proof. In one direction, suppose there exists a witness graph $f = f^S \boxtimes f^T$. By Lemma 5, $\forall \sigma^T \in \mathit{Cmp}(f^T)$, $\exists \sigma \in \mathit{Cmp}(f) : \sigma^T \sim_{st} \sigma{\uparrow}_T$. By Lemma 5, $\exists \sigma^S \in \mathit{Cmp}(f^S) \, : \, \sigma^S \sim_{st} \sigma{\uparrow}_S$. By Definition 6, $o(\sigma{\uparrow}_T) = o(\sigma{\uparrow}_S)$; so we conclude $\sigma^T \sim_{st} \sigma^S$. Similarly, we prove $\forall \sigma^S \in \mathit{Cmp}(f^S)$, $\exists \sigma^T \in \mathit{Cmp}(f^T) \, : \, \sigma^S \sim_{st} \sigma^T$. Thus, f^T is a correct translation of f^S.

In the other direction, suppose, f^T is a correct translation of f^S. There exists a witness comparison graph $f = f^S \boxtimes f^T$. This can be shown by construction of such graph while observing the computations of f^S and f^T. The graph will be finite, the construction is not finite due to possible infinite input computations.

To prove the second statement, suppose again that f^T is a correct translation of f^S. We need to show that there does not exist a comparison graph which is not a witness of correct translation. Let's proof this by contrapositive. Assume there exists f such that $\exists \sigma \in \mathit{Cmp}(f)\ :\ o(\sigma{\uparrow_S}) \neq o(\sigma{\uparrow_S})$, implying $\neg(\sigma{\uparrow_S} {\sim_{st}}\ \sigma{\uparrow_S})$. By Lemma 5, $\exists \sigma^S\ \in \mathit{Cmp}(f^S) :\ \sigma^S \sim_{st} \sigma{\uparrow_S}$ and $\exists \sigma^T\ \in \mathit{Cmp}(f^T) :\ \sigma^T \sim_{st} \sigma{\uparrow_T}$ but by our assumption, $\neg(\sigma^S\ \sim_{st}\ \sigma^T)$. Since f^S is a correct translation of f^T, $\exists \varrho^T \in \mathit{Cmp}(f^T)\ :\ (\varrho^T \sim_{st} \sigma^S)\ \wedge\ (\varrho^T \neq \sigma^T)$. By definition of composed transition, σ^T and σ^S agree on all input and read values. On the other hand, $\varrho^T\ \sim_{st}\ \sigma^T$ implies that ϱ^T and σ^T agree on all input and read values. However, since f^T is deterministic, $\varrho^T = \sigma^T$; hence we reach a contradiction.

□

Lemma 2 *No spurious predictions are possible:* if the match (e^S, e^T) is made with φ_n^k, it also complies with φ_n^{fix}. As a practical consequence, algorithm `compose` never has to backtrack (due to removal of previously added edges).

Proof. Suppose, a match is made using φ_n^k on iteration $k+1$ and $(\varphi_n^k\ \wedge\ c^S\ \wedge\ c^T)$ is satisfiable. Since $\varphi_n^k\ =\ (\varphi_n^{fix} \wedge \Phi_n^k)$, assertion $(\varphi_n^{fix}\ \wedge\ c^S\ \wedge\ c^T)$ is also satisfiable. Thus, the match would also be made with φ_n^{fix}. □

Theorem 2 *(Termination) Algorithm* `compose` *terminates.*

Proof. Following Lemma 2, the algorithm never removes edges and nodes from the comparison graph, so the graph monotonically grows. In addition, each cross-product edge and node can only be added once, so the number of new nodes and edges is bounded. $\square$

Theorem 2 *(Soundness) If algorithm* `compose` *succeeds, the resulting graph* $f = f^S \boxtimes f^T$ *satisfies all of the requirements presented in Section 2.4.2.*

Proof. Clearly, the structural requirement, Rule 1, is satisfied. Since we explicitly require that the graph does not include any ϵ-cycles, Rule 2 is also satisfied.

To prove Rule 3, let's show that all computations of f^S and f^T are covered by the constructed graph. To support the statement, we argue that if a source or target edge is never added to the comparison graph, it does not participate in any computations of the input system.

Suppose a target edge e^T from n^T to m^T, conditional on c^T, is the first edge not added to the fully constructed comparison graph (assume topological ordering). Since node $n = \langle n^T, n^S \rangle$ belongs to the comparison system, it had been added to the `WorkList`; and every time we attempted to match the source edges outgoing from n^S with e^T, we failed. Let's consider the last time the matching was attempted. Since this was the last try, no new edges that may influence the states at location n were added afterwards, so the invariant used $\varphi_n^k = \varphi_n^{fix}$. The match failed, meaning that for every source edge e_i^S outgoing form n^S and conditioned on c_i^S, $\varphi_n^{fix} \wedge c_i^S \wedge c^T$ was unsatisfiable. Since the source program is non-blocking: $\wedge c_i^S = true$, it must be that $\varphi_n^{fix} \wedge c^T$ is unsatisfiable. Recall that e^T is the first edge of the target program that is not added to the comparison program, so the invariant of the comparison system at location n, φ_n^{fix}, fairly represents the states of the target program; thus it must be that c^T evaluates to false in all computations of the target, and the transitions corresponding to e^T are never taken. An inductive argument can be used for all the following edges. $\square$

Theorem 2 *Completeness* If f^T and f^S are consonant, algorithm `compose` succeeds in construction of a comparison graph $f = f^S \boxtimes f^T$ given a strong enough *InvGen*.

Proof. Algorithm `compose` fails only when we cannot match branches of the source and target. Consider the `matchEdges` rules presented in Section 2.4.3. For the first two conditions not to be applicable, it must be the case that the algorithm visits a node $\langle n^S, n^T \rangle$ such that $(n^S \in \mathcal{P}^S) \wedge (n^T \in \mathcal{P}^T) \wedge (\tau(n^S) \neq \tau(n^T))$. Our goal is to show that this situation is ruled out. In particular, we are going to show that $(n^S \in \mathcal{P}^S) \wedge (n^T \in \mathcal{P}^T) \implies \tau(n^S) = \tau(n^T)$. (Recall that $\mathcal{P}^T$ and $\mathcal{P}^S$ denote the sets of the source and the target cut points. $\tau(n)$ denotes the type of a node. See Section 2.2.4 for the full definition of these concepts.)

Since *InvGen* is strong enough, the node $\langle n^S, n^T \rangle$ is reachable in f. Thus, there exists $\sigma^C \in Cmp(f)$ that goes through the node $\langle n^S, n^T \rangle$. Applying the same argument as for the soundness proof to the partially constructed graph and rules Rule 1 and Rule 2, there exist $\sigma^S \in Cmp(f^S)$ and $\sigma^T \in Cmp(f^T)$ that differ from $\sigma{\uparrow}_S$ and $\sigma{\uparrow}_T$ by only finite number of ϵ-transitions. Since the composed programs are consonant it only remains to show that n^S and n^T are the i^{th} cut points in the cut point sequences for σ^S and σ^T.

Consider an arbitrary node $\langle \hat{n}^S, \hat{n}^T \rangle$ that is visited no later then the node $\langle n^S, n^T \rangle$. Let *rank* of either a source or a target node $\hat{n}$ be the number of cut points before or at the node the corresponding computation (σ^S or σ^T, respectively). We are going to use induction on the length of σ^C to proof that rank of $\hat{n}^S$ is the same as rank of $\hat{n}^T$ when both are cut points or when both are not. If only one of the systems is at a cut point, its rank is precisely one more then that of the other one.

- **Base Case**: When $\langle \hat{n}^S, \hat{n}^T \rangle$ is the first node of σ^C, $\hat{n}^S$ and $\hat{n}^T$ are the roots of the composed procedures and are the first elements in the corresponding cut point sequences. Therefore, the rank of both nodes is simply 1.

- **Induction Step**: According to `matchEdges` there are only two ways to extend the computation:

  - **Matching edges of the same type**: For this rule to apply both systems or none must be at a cut point location and by induction hypothesis have the same rank i. When both systems move to a new cut point their ranks are simply increased by 1. When both system move to a non cut point node (unconditional assignment node) the rank is preserved. And, finally, if only one of system moves to a new cut point its rank becomes $i + 1$, an the other one must still have the rank i.

  - **Adding ϵ-transitions**: For this rule to apply and according to our induction hypothesis one of the systems must be at cut point and have a rank i. The other system should be about to preform an assignment and have rank $i - 1$. These ranks and node types are either preserved by this match rule or the slower system catches up and both systems reach their i^{th} cut point.

$\square$

Appendix B

Correctness of SPOR

In this section, we present a formal proof for the correctness of superstep partial order reduction (SPOR), specifically, we will show that the systems M and M' defined in section Section 3.5.2 are stuttering equivalent. To do this, we use a proof technique similar to the one presented in [15].

Given an atomic proposition p and a transition α, let $\mathcal{V}(p, \alpha) \mapsto (S_p^\alpha \subset S)$ be a function that returns the set of states S_p^α such that $(s \in S_p^\alpha) \longleftrightarrow (p(s) \neq p(\alpha(s)))$. Thus, $\mathcal{V}(p, \alpha)$ partitions the sate space into the states in which the execution of α flips the value of p and those in which the execution of α does not change the value of p.

Lemma 6.

$\forall p \in AP \ : \ (s \notin \mathcal{V}(p,t)) \ \wedge \ I(t',t) \implies$

$(t'(s) \notin \mathcal{V}(p,t)) \ \wedge \ ((t(s) \in \mathcal{V}(p,t')) \longleftrightarrow (s \in \mathcal{V}(p,t')))$

In other words, given a path $s \xrightarrow{t} t(s) \xrightarrow{t'}$ such that t and t' are independent and t

is not visible from s, the invisible transition can be moved to the right resulting in a path $s \xrightarrow{t'} t'(s) \xrightarrow{t}$ and the visibility and invisibility of the transitions with respect to every predicate is preserved.

Proof. Let the proposition p be $(x = val)$ for some variable x and value val. First, if the invisible transition t does not modify x, the lemma trivially holds. Second, consider the case when t does modify x. Then since $I(t, t')$, both t and t' set x to the same value val'. Without the loss of generality, assume that $p(s) = true$ (p holds at s). That means that x evaluates to val in s. Two cases are possible. If $val' = val$, t' is invisible at s and $t(s)$, and t is invisible at s and $t'(s)$, so the invisibility property is preserved. If $val' \neq val$, then t is visible at s, which contradicts the lemma's assumption.

$\square$

Lemma 3 (from Section 3.5.2) *If Λ is a core of Θ at s, then*

1. $\Theta(s) = (\Lambda \circ (\Theta \setminus \Lambda))(s)$
2. $\Pi(s,\ \Theta) \sim_{st} \Pi(s,\ \Lambda) \sim_{st} \Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$

Proof. Lemma 3.1 follows directly from Definition 9.1. Let's prove Lemma 3.2. By Lemma 3.1, path $\Pi(s,\ \Theta)$ is enabled if and only if $\Pi(s,\ \Lambda)$ and $\Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$ are enabled. Path $\Pi(s,\ \Lambda)$ is obtained from $\Pi(s,\ \Theta)$ by moving the transitions from $\Theta \setminus \Lambda$ to the right of the transitions in Λ. Since the transitions in $\Theta \setminus \Lambda$ are invisible from the corresponding states, by Lemma 6, the move preserves the visibility of all the transitions. Thus, if there is one visible transition t in $\Pi(s,\ \Theta)$, it will be the only one visible transition in both $\Pi(s,\ \Lambda)$ and $\Pi(s,\ \Lambda \circ (\Theta \setminus \Lambda))$. By the definition of visibility, t is the only transition that may change the label of the states along the paths. In addition, the paths are finite and the very first and the very last state of all the paths must be the same by Lemma 3.1, implying that they have the same label. Consequently, the paths must be stuttering equivalent. $\square$

Let ω be a path of M starting at some initial state s_0. First, we will iteratively construct a sequence of strings $tr(\omega) = \pi_0,\ \pi_1,\ \pi_2,\ \dots$. Every i^{th} sequence can be partitioned as following: $\pi_i = \eta_i \circ o_i$, where $|\eta_i| = i$; η_i is a string of superstep transitions; and o_i is a sequence of transitions of the input system M. Furthermore, $en_{M'}(s_0, \eta_i)$ - there exists a path in M' starting at s_0 with transitions η_i. Let $s_i = \eta_i(s_0)$ be the final state on that path, then $en_M(s_i, o_i)$ - there exists a path in M starting at s_i with transitions o_i.

Suppose that we have partially constructed the sequence $\pi_0, \pi_1, ..\pi_i$ for some i. We construct π_{i+1} as follows. Let $o_i = \theta_i \circ \rho_i$, such that $|vis(\theta_i)| \leq 1$. Let $\lambda_i \in Cap(s_i)$ be a core of θ_i, which must exist as a consequence of Definition 9 and Definition 10. We choose $\eta_{i+1} = \eta_i \circ s_i \rhd \lambda_i$ and $o_{i+1} = (\theta_i \setminus \lambda_i) \circ \rho_i$. Then $s_{i+1} = \eta_{i+1}(s_0)$.

Since $s_i \rhd \lambda_i \in \overline{Cap}$, $en_{M'}(s_0, \eta_{i+1})$. Let's show that $en_M(s_{i+1}, o_{i+1})$. By inductive assumption, $en_M(s_i, \theta_i \circ \rho_i)$. By Lemma 3.1, $en_M(s_i, (\lambda_i \circ (\theta_i \setminus \lambda_i)) \circ \rho_i)$. By Definition 8, $s_{i+1} = \lambda_i(s_i)$, so $en_M(s_{i+1}, (\theta_i \setminus \lambda_i) \circ \rho_i)$, which was required to show.

Lemma 7. *Let η be the string constructed as the limit of the finite strings η_i. Then, $\Pi(s_0,\ \eta)$, belongs to the reduced transition graph M'.*

Proof. String η is well defined since $\forall i\ :\ \eta_i < \eta_{i+1}$. Additionally, for all i: $en_{M'}(s_0, \eta_i)$ implying that $\Pi(s_0,\ \eta_i)$ is a path in M'. $\square$

Definition 13. *Given two transitions t and t' we are going to use notation $t \cong t'$, if and only if either $t = t'$ or $(t = s \rhd \Lambda) \ \wedge \ (vis(\Pi(s, \ \Lambda)) = t')$.*

Lemma 8. *The following hold for all i, j such that $j \geq i \geq 0$.*

1. *$vis(\pi_i) \cong vis(\pi_j)$.*

2. *Let ξ_i be a prefix of π_i and ξ_j be a prefix of π_j such that $vis(\xi_i) \cong vis(\xi_j)$. Then $L(\xi_i(s_0)) = L(\xi_j(s_0))$.*

Proof. It is sufficient to consider the case where $j = i+1$. Then $\pi_i = \eta_i \circ \theta_i \circ \rho_i$ and $\pi_j = \eta_i \circ s \rhd \lambda_i \circ (\theta_i \setminus \lambda_i) \circ \rho_i$.

To prove Lemma 8.1, it is sufficient to show that $vis(\theta_i) \cong vis(s \rhd \lambda_i \circ (\theta_i \setminus \lambda_i))$. By Definition 9 and Lemma 6, transitions in $(\theta_i \setminus \lambda_i)$ are invisible; furthermore, at most one transition in λ_i and, consequently, one transition in θ_i is visible. Let $vis(\theta_i) = vis(\lambda_i) = t$, then $t \cong s \rhd \lambda_i$. On the other hand, if all of the transitions in λ_i are invisible, then the equality trivially holds.

To prove Lemma 8.2, suppose $vis(\theta_i) \cong s_i \rhd \lambda_i \cong t_k$ and then $\pi_i = \eta_i \circ t_{i+1} \circ .. \circ t_k \circ .. \circ t_l \circ \rho_i$.

Lemma 8.2 holds for prefixes with $k-1$ visible transitions. It is enough to show that $\forall t_j : i < j < k \ : \ L(\eta_i(s_0)) = L(\eta_i \circ t_{i+1} .. \circ t_j(s_0))$ holds; which is the case since all t_j are invisible.

Next, we show that the statement holds for the shortest prefixes with k visible transitions. Let $s' = \eta_i \circ t_{i+1} \circ .. \circ t_k(s_0)$ - the first state on the path π_i after taking the visible transition t_k; and let $s''' = \eta_i \circ s \rhd \lambda_i(s_0)$ - be the first state after the superstep transition is taken. Path π_j has been obtained after application of the superstep reduction to a path $\pi_i' = \eta_i \circ (\gamma \circ t_k \circ \alpha) \circ (\theta_i \setminus \lambda_i) \circ \rho_i$, where $\lambda_i = \gamma \circ t_k \circ \alpha$. Let s'' be the first state on this path after transition t_k: $s'' = \eta_i \circ \gamma \circ t_k(s_0)$. Since λ_i is a core of θ_i, $s' = \beta(s'')$, where $\beta = (t_{i+1} \circ .. \circ t_{k-1}) \setminus \gamma$ is a sequence of invisible transitions independent of transitions in λ_i. Additionally, $s''' = \alpha(s'')$, furthermore, α contains no visible transitions since $|vis(\lambda_i)| \leq 1$. By definition of invisible transitions, $L(s') = L(s'')$ and $L(s''') = L(s'')$ implying $L(s') = L(s''')$.

The statement also holds for longer prefixes ξ_i and ξ_j, up to $\xi_i = \eta_i \circ \theta_i$ and $\xi_j = \eta_i \circ s \rhd \lambda_i \circ (\theta_i \setminus \lambda_i)$, since the invisible transitions preserve the state labels. Finally, let $s' = \eta_i \circ \theta_i(s_i) = (s \rhd \lambda_i \circ (\theta_i \setminus \lambda_i))(s_i)$. All prefixes that contain more visible transitions then those just considered will end in the same state since the next visible transition should be on the path $\Pi(s',\ \rho_i)$, which is shared by both $\Pi(s_0,\ \pi_i)$ and $\Pi(s_0,\ \pi_j)$. The existence of the shared state s' is guaranteed by Lemma 3.1. The same argument is used when $|vis(\theta_i)| = 0$.

□

Lemma 9. *Let v be a prefix of $vis(\pi_0)$. Then there exists a string η_i such that $v \cong vis(\eta_i)$.*

Proof. The proof is by induction on the length of v. For $|v| = 0$, the empty string η_0 satisfies the requirement. In the inductive step we must show that for a visible transition t, if $v \circ t$ is a prefix of $vis(\pi_0)$ and $\exists\ \eta_i : v \cong vis(\eta_i)$, then there is a string η_j with $j > i$ such that $vis(\eta_j) \cong v \circ t$. First, consider the case when t is the visible transition of λ_{i+1} then the statement holds with $j = i + 1$. Next, consider the case when λ_{i+1} does not contain a visible transition. Then, t is the first visible transition in o_{i+1}. Let δ be a prefix of o_{i+1} such that $o_{i+1} = \delta \circ t \circ \chi$ and

δ does not contain t. Then, $\forall\ k : (i+1 < k)\ \wedge\ vis(\lambda_k) \neq t$, λ_k is a non-empty (noncontiguous) subsequence of δ. Thus, at some point δ must be exhausted and $\exists\ j : k < j\ \wedge\ vis(\lambda_j) = t$. □

Theorem 3 (from Section 3.5.2) *The transition graphs M and M' are stuttering equivalent.*

Proof. By Definition 11, M and M' have the same set of initial states.

In one direction, we need to show that for each path η in M', there exists a path σ in M such that $\eta \sim_{st} \sigma$. Let $\eta = s_0 \overset{s_0 \rhd \lambda_0}{\longrightarrow} s_1 \overset{s_1 \rhd \lambda_1}{\longrightarrow} ...$ Consider path $\sigma = s_0 \xrightarrow{\lambda_0} s_1 \xrightarrow{\lambda_1} ...$ By Definition 10, each string λ_i is enabled at the corresponding state s_i, and, thus, σ belongs to M. Further, by Definition 9, $|vis(\lambda_i)| \leq 1$ implying that $\eta \sim_{st} \sigma$.

In the other direction, we need to show that for each path σ in M, there exists a path η in M' such that $\sigma \sim_{st} \eta$. Indeed, we will show that the path η constructed as the limit of the finite path η_i is stuttering equivalent to $\sigma = \pi_0$.

First, we show that $vis(\pi_0) \cong vis(\eta)$. By Lemma 9, η contains the visible transitions equivalent to those in π_0 in the same order, because for any prefix of π_0, there is a prefix η_i of η with equivalent visible transitions. On the other hand, π_0 must contain the visible transitions of η in the same order. Take any prefix η_i of η. According to *Lemma* 8.1, $\pi_i = \eta_i \circ o_i$ has the same visible transitions as π_0. Thus, π_0 has a prefix with the same sequence of visible transitions as η_i.

Let $\pi_0 = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} ...$ and $\eta = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} ....$ We construct two infinite sequences of indexes $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ that define the corresponding stuttering blocks of σ and η. Assume that both π_0 and η contain at least n visible transitions. Let i_n be the length of the smallest prefix ξ_{i_n} that contains exactly n visible transitions. Let j_n be the length of the smallest prefix η_{j_n} such that $vis(\xi_{i_n}) \cong vis(\eta_{j_n})$. By Lemma 8.2, $L(s_{i_n}) = L(r_{j_n})$. In addition, let $\xi_{i_{n-1}}$ be the smallest prefix of σ with $n-1$ visible transitions, and similarly define $\eta_{j_{n-1}}$. As before, let $|\xi_{i_{n-1}}| = i_{n-1}$ and $|\eta_{j_{n-1}}| = j_{n-1}$. Then for $i_{n-1} \leq k < i_n - 1 \,:\, L(s_k) = L(s_{i_{n-1}})$ since the transitions between i_{n-1} and $i_n - 1$ are invisible. Similarly, for $j_{n-1} \leq l < j_n - 1 \,:\, L(r_l) = L(r_{j_{n-1}})$.

If both π_0 and η have infinitely many visible transitions, then this process will construct two infinite sequences of indexes. In the case where π_0 and η contain only a finite number m of visible transitions, we have that for $k > i_m$, $L(s_k) = L(s_{i_m})$ and for $l > j_m$, $L(r_l) = L(r_{j_m})$. We then set for $k \geq m$, $i_{k+1} = i_k + 1$ and for $l \geq m$, $j_{l+1} = j_l + 1$. Thus, for $k \geq 0$, the blocks of states $s_{i_k}, s_{i_k+1}, ..., s_{i_{k+1}-1}$ and $r_{i_k}, r_{i_k+1}, ..., r_{i_{k+1}-1}$ are the corresponding stuttering blocks; implying $\pi_0 \sim_{st} \eta$.

□

Lemma 4 (from Section 3.5.3) *For any state s, the set $ThreadedCap(s)$ covers s.*

Proof. Consider a path $\sigma \in \mathcal{E}(M,\ s)$ that results from execution of a finite number of threads in some order. We will show that the set $ThreadedCap(s)$ contains a string Λ such that Λ is a core of a finite prefix of σ. Let W denote the set containing only the very last transitions from each superstep string in $ThreadedCap(s)$, which is defined in Definition 12. Let Θ be a finite prefix of $tr(\sigma)$ such that all its transitions except for the last one are not in W.

Such prefix exists. Suppose that is not the case. Since the supersteps are finite, there must exist a transition on the path σ that is not contributed by a superstep. Suppose, the very fist such transition belongs to thread i and it is the l^{th} transition of σ. Let $\Lambda_i^s = \alpha_1\alpha_2..\alpha_k\alpha_{k+1}..\alpha_m$ and let η be the prefix of $\sigma : |\eta| = (l-1)$. Let α_k be the last transition of Λ_i^s in η. Then α_{k+1} has been disabled by a transition that occurs before it in η. It cannot be a transition of thread i since $en_M(s, (\Lambda_i^s))$. So this transition has to belong to a superstep of another thread, which contradicts the independence requirement.

Without the loss of generality, assume that the last transition of Θ is a transition of thread i. Θ contains all the transitions of Λ_i^s. Every other transition $t \in \Theta \setminus \Lambda_i^s$ belongs to a superstep of another thread and $t \notin W$. So by the independence requirement, transitions in $\Theta \setminus \Lambda_i^s$ are independent of the transitions in Λ_i^s. In addition, there might be at most one visible transition and $vis(\Theta) = vis(\Lambda_i^s)$. Thus, Λ_i^s is a core of Θ at s. $\square$

Bibliography

[1] CVC3: An Automatic Theorem Prover for Satisfiability Modulo Theories (SMT). `http://www.cs.nyu.edu/acsys/cvc3/`.

[2] Open Research Compiler for Itanium^{TM} Processor Family. `http://ipf-orc.sourceforge.net`.

[3] Peterson's algorithm. `http://en.wikipedia.org/wiki/Peterson's_algorithm`.

[4] The CompCert verified compiler. `http://compcert.inria.fr/`.

[5] The Coq proof assistant. `http://coq.inria.fr/`.

[6] The GNU Compiler Collection. `http://gcc.gnu.org`.

[7] The LLVM Compiler Infrastructure Project. `http://llvm.org`.

[8] The Spin Model Checker. `http://spinroot.com/spin/whatispin.html`.

[9] The YICES SMT Solver. `http://yices.csl.sri.com`.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[11] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *In VSTTE*, page 1603, 2005.

[12] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[13] Gilles Barthe, Pedro D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop*, page 100. IEEE Computer Society, 2004.

[14] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[15] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[16] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[18] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[19] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.

[20] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[21] Yi Fang. *Translation Validation of Optimizing Compilers*. PhD thesis, New York University, 2005.

[22] Yi Fang and Lenore D. Zuck. Improved invariant generation for TVOC. In *Proceedings of the* 5^{th} *International Workshop on Compiler Optimization meets Compiler Verificaiton*, 2006.

[23] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM Symposium on Programming Languages (POPL)*, pages 110–121, 2005.

[24] Robert W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19:19-32, 1967.

[25] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[26] Alex Groce and Rajeev Joshi. Extending model checking with dynamic analysis. In *Conference on Verification, Model Checking and Abstract Interpretation*, 2008.

[27] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In *SPIN Workshop on Model Checking of Software*, pages 95–112, 2007.

[28] Sumit Gulwani and George Necula. Global value numbering using random interpretation. In *31st Symposium on Principles of Programming Languages*, pages 342–352. ACM Press, 2004.

[29] Sumit Gulwani and Ashish Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *The 15th European Symposium on Programming*, pages 279–293. Springer, March 2006.

[30] Daniel Halperin, Thomas S. Heydt-Benjamin, Benjamin Ransford, Shane S. Clark, Benessa Defend, Will Morgan, Kevin Fu, Tadayoshi Kohno, and William H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. IEEE Symposium on Security and Privacy, 2008.

[31] G. J. Holzmann and D. Peled. An improvement in formal verification. In *International Conference on Formal Methods for Networked and Distributed Systems*, 1994.

[32] Gerard Holzmann and Margaret Smith. A practical method for verifying event-driven software. In *International Conference on Software Engineering*, pages 597–607, 1999.

[33] Gerard J. Holzmann. An analysis of bitstate hashing. Formal Methods in Systems Design.

[34] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[35] Gerard J. Holzmann and Dragan Bosnacki. The design of a multi-core extension of the spin model checker. In *IEEE Transactions on Software Engineering*, volume 33, pages 659–674, October 2007.

[36] Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In *SPIN Workshop on Model Checking of Software*, pages 76–91, 2004.

[37] Gerard J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *SPIN Workshop on Model Checking of Software*, pages 23–32, 1996.

[38] G.J. Holzmann and D. Bosnacki. The design of a multi-core extension of the Spin model checker. *IEEE Transactions on Software Engineering*, 33:659–674, 2007.

[39] Ying Hu, Clark Barrett, Benjamin Goldberg, and Amir Pnueli. Validating more loop optimizations. In *Proceedings of the 4^{th} International Workshop on Compiler Optimization meets Compiler Verificaiton*, 2005.

[40] Yuqiang Huang, Bruce R. Childer, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis Symposium*, pages 281–300. Springer, 2006.

[41] Anick Jesdanun. GE energy acknowledges blackout bug. Associated Press, 2004. http://www.securityfocus.com.

[42] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[43] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 220–231, 2003.

[44] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32th ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[45] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of the 33th ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.

[46] Nancy G. Leveson. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993. http://sunnyday.mit.edu/papers/therac.pdf.

[47] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[48] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 34th ACM Symposium on Programming Languages (POPL)*, pages 446–455, 2007.

[49] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *Symposium on Operating System Design and Implementation*, 2002.

[50] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2008.

[51] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation*, pages 83–95. ACM Press, 2000.

[52] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th Conference on Computer Aided Verification*, pages 409–423. Springer, 1993.

[53] A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. *Lecture Notes in Computer Science*, 1443:235–250, 1998.

[54] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[55] Amir Pnueli. Verification of procedural programs. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, pages 543–590. College Publications, 2005.

[56] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, April 2005.

[57] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proc. of the Sixth International Joint Conference CAAP/FASE*, Aarhus, Denmark, 1995.

[58] Martin C. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, MIT, 1999.

[59] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st Symposium on Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

[60] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 323–335, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[61] Loren Taylor Simpson. *Value-Driven Redundancy Elimination.* PhD thesis, Rice University, 1996.

[62] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2008.

[63] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of Lazy Code Motion. In *Programming Language Design and Implementation 2009*, 2009. To appear.

[64] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2009.

[65] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[66] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2007. Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings.

[67] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for the translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

[68] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop tranformations. *Formal Methods in System Design*, 27(3):335–360, 2005.

[69] Lenore Zuck, Amir Pnueli, and Raia Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann Institute of Science, 2001.