Parallel Proofs for Parallel Programs

by

Zachary Kincaid

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Parallel Proofs for Parallel Programs

Zachary Kincaid
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

This dissertation addresses the problem of automated reasoning about multi-threaded programs. Multi-threaded programs are notoriously prone to error, making them an attractive target for formal methods which can be used to *guarantee* that programs satisfy desirable properties. *Automated* formal methods lift the burden of reasoning about concurrency from software developers onto the shoulders of machines. However, concurrency raises some significant algorithmic challenges for formal methods, particularly in reasoning about complex interactions between threads. This dissertation aims to tame this problem by developing *parallel* logical foundations for multi-threaded programs.

Classical approaches to reasoning about multi-threaded programs are based on developing clever ways to reason about them as if they were sequential. There are two prototypical examples of this approach: the *interleaving model* and the *thread-modular model*. In the interleaving model, a multi-threaded program is *compiled* into a sequential program that interleaves the behaviour of all threads, after which we may reason about the program using classical techniques for sequential programs. In the thread-modular model, we reason about each thread independently as a sequential program, subject to side-conditions that enforce that the reasoning about each thread is robust under interference from the others.

This dissertation takes a different tack: rather than devise a way to apply sequential reasoning to multi-threaded programs, we develop foundations for reasoning about parallelism directly. On these foundations, we build static analysis and software model checking algorithms for verifying and refuting properties of multi-threaded programs. We show that direct and explicit representation of parallelism enables tractable and precise automated reasoning.

To Leah

Acknowledgements

I had a wonderful time in my Ph.D. I was not in a great hurry to graduate. I have many people to thank (or perhaps, to blame) for this.

First I would like to thank my advisor, Azadeh Farzan. Azadeh has been tremendously supportive throughout my studies, and has opened countless doors for me in the research community. She taught me how to write and that it doesn't matter how good a paper is if no one reads it. Azadeh gave me a great deal of latitude to explore while also being deeply involved in my research. I am enormously grateful to have had her as my advisor.

I was fortunate to have supportive and kind committee members in Marsha Chechik and Eric Hehner. My thanks also go to Sheila McIlraith and Ranjit Jhala for taking the time to review my thesis and provide helpful feedback.

I am grateful that I had the opportunity to collaborate with some brilliant researchers over the past several years: Aws Albarghouthi, Josh Berdine, Swarat Chaudhuri, Marsha Chechik, Byron Cook, Azadeh Farzan, Sumit Gulwani, Arie Gurfinkel, Masami Ito, Lila Kari, Yi Li, Andreas Podelski, and Shinnosuke Seki. Thank you for doing all the work.

Andreas Podelski is a co-author of much of the work that went into this thesis. I feel very fortunate to have had Andreas as a mentor and collaborator. Andreas taught me the importance of distilling a research project down to that *one* essential idea. He often reminds me that one should not take things so seriously – research is supposed to be fun!

My thanks go to Swarat Chaudhuri for being a great advocate. Working with Swarat is always stimulating – his enthusiasm for research is infectious.

I'd like to thank my cohort in the SE group at UofT: Aws Albarghouthi, Michalis Famelis, and Alicia Grubb. I have had the pleasure of working with Aws on several occasions, and I hope there are many more to come. He is a great research partner and a great friend. I'd also like to thank Rick Salay for the years of interesting discussions following our joint curiosity. It was always a welcome break from research to explore new topics with Rick.

I learned a lot from working with Byron Cook and Josh Berdine at Microsoft Research Cambridge. Thank you Byron for hosting my stay, and thank you Josh for lending your tremendous brain to our project.

Lila Kari gave me my first opportunity to do research. My experience with Lila and Shinnosuke Seki convinced me that research was something I wanted in my future. I'd like to thank Tim Teitelbaum and Michael McDougall for responding to an unsolicited email from a second year undergrad looking for a summer job. My exposure to static analysis at GrammaTech was a formative experience. It was at GrammaTech that I first learned about program dependence graphs, which inspired my interest in data-flow based program representations.

I'd like to thank my family – Mom, Dad, Stefan, Brianna, and Will – for being with me along every step of my academic journey. Thank you for your encouragement and always showing an interest in my research (particularly, asking "have you finished your thesis yet?") Finally, the answer is yes.

To my wife Leah, I dedicate this thesis. Thank you for being my partner in all things. Thank you for celebrating with me when my research is going well and encouraging me when it isn't. Thank you for making sure that I survive paper deadlines. Thank you for everything.

Contents

Part I

Prologue

Chapter 1

Introduction

The demand for reliable, efficient, and secure multi-threaded software is large and it is growing. Some problems, for example those that involve interacting with physical systems, intrinsically require multi-threaded solutions. Even for problems that do not *require* multi-threading, it is often desirable to use multi-threading to reap the benefits of multi-core processors. Despite its prominence in modern software, multi-threading remains a serious obstacle to developing reliable software: faults due to concurrency are difficult to find, to debug, and even to reproduce. One promising direction is to look to automation for help in meeting demand for reliable multi-threaded software.

The field of program analysis is concerned with developing algorithmic techniques to answer questions about program behaviour, such as *can an expression be safely hoisted out of a loop?*, or *what are the possible values of a given variable at a given point?*, or *is it possible for a given buffer access to overflow*? The most widespread use of program analysis is in compilers, where program analysis is used to prove safety of optimizations and to prove absence of certain types of bugs such as type errors and uninitialized variables. Automated verification, in which algorithms prove that programs satisfy correctness properties, is an active area of research that has also begun to enjoy success in industry. The great promise of automated verification is that some of the effort of ensuring reliability of software can be delegated to a machine.

Unfortunately, the same things that make it difficult to build and test multi-threaded software make it difficult to analyze algorithmically. As a result, techniques for analyzing multi-threaded programs are relatively less developed than techniques for sequential programs. This dissertation takes a step forwards in this direction by developing algorithmic techniques for *finding* and *proving the absence of* faults in multi-threaded software.

1.1 Automated Verification

In this dissertation, we are interested in algorithms for answering verification problems of the form *do all executions of the program satisfy some property X?* For example, *do all executions to a given point end in a state where a given assertion holds*?, or *does there exist an execution that ends at a given point?* (equivalently, *do all executions avoid a given point?*) Any such property can be either *verified* – shown to hold on every execution, or *refuted* – shown to *not* hold in *some* execution.

It is an easy corollary of the undecidability of the halting problem [Turing, 1936] that there is no

terminating algorithm that, given an arbitrary program and a verification problem, either verifies the property or refutes it. Worse still, we cannot even hope that there is an interesting sub-class of program properties that are decidable, because the halting problem can be reduced to any non-trivial decision problem regarding program behaviour [Rice, 1953]. To make progress on the goal of automated verification, we must relax what we expect of a verification algorithm.

Suppose we have a hypothetical procedure $A(P, \varphi)$ that, given a program P and a property φ, returns either "yes" (P satisfies φ) or "no" (it does not). We may call A an *automated verification procedure* if it is *sound for verification*: whenever $A(P, \varphi)$ returns "yes", then P satisfies φ. Clearly, this is an obtainable desideratum: a procedure that simply returns "no" for every input is (even if it is not particularly useful) sound for verification. This trivial procedure returns the correct answer when P does not satisfy φ, but the wrong answer when it does – such a wrong answer is called a *false alarm*. If A does *not* produce false alarms, it is called *sound for refutation*. A trivial argument (as the one above) shows that refutation-soundness is obtainable. In fact, it is possible to obtain procedures that are *complete* for refutation: if P fails to satisfy φ, then $A(P, \varphi)$ returns "no." However, the fact that the verification problem is undecidable precludes any terminating procedure from being sound and complete for refutation.

There are two schools of automated verification: *static analysis* and *software model checking*. The essential difference between the two is which desiderata is relaxed:

- Static analysis encompasses techniques that are terminating and sound for verification, but unsound for refutation. Compilers make heavy use of static analysis – for example, to answer queries of the form *is it safe to perform this optimization?* Static analysis is used in situations where false alarms can be tolerated (e.g., the compiler simply fails to perform a safe optimization), but non-termination cannot be (a sub-optimal program is better than one that does not compile).

  Static analysis is also the basis of several commercial bug-finding tools, where false alarms are poorly tolerated, but this concern is trumped by efficiency and scalability considerations ("If the tool can't check a system, file, code path, or given property, then it won't find bugs in it." [Bessey et al., 2010]).

- Software model checking encompasses techniques that are sound for verification and refutation, but that do not necessarily terminate. A salient advantage of software model checkers is that they do not suffer from false alarms. False alarms are problematic because they require manual effort to classify bug reports as either real bugs or false alarms. The practical result is that bug reports are often mis-classified and subsequently ignored: [Dillig et al., 2012] reports a user study in which *over 50%* of bug reports were mis-classified, and [Bessey et al., 2010] reports a number of anecdotes from experience employing static analysis commercially.

  Another advantage of software model checking techniques is that, should a property fail to hold, they are often able to produce a *counter-example* that witnesses its failure. Such a counter-example can be valuable for diagnosing the source of the fault.

Each school of automated verification has its associated strengths and weaknesses and there are settings in which one or the other is more appropriate. Moreover, the schools can benefit from cross-fertilization of ideas. This dissertation makes contributions to both schools.

1.2 The Problem with Concurrency

Multi-threaded programs are an attractive target for automated verification, in part because they are poorly suited to testing, the conventional method for increasing reliability of software systems. The problem with testing concurrent programs is that the scheduler may (in principle) interleave thread executions in an arbitrary order, yielding highly nondeterministic behaviour. As a result, the success of finding a bug in a multi-threaded program using testing relies not only on being able to find the right inputs, but also getting the right schedule (which the test designer has little control over). Even if testing successfully uncovers a bug, it may be difficult to reproduce because running the program again with the same input may yield different behaviour. Such bugs may also be difficult to debug, since attaching a debugger to a process affects scheduler behaviour, and may even cause bugs to disappear (earning these bugs the moniker *Heisenbugs*).

Automated verification is a promising way to increase the reliability of multi-threaded programs because it is not subject to the whims of a scheduler: a verifier accounts for all possible behaviours of the scheduler and always gives consistent answer. However, the theoretical appeal of using an automated verifier for multi-threaded programs is diminished by the practical reality of building one. The (already undecidable) verification problem is complicated by the fact that the verifier must account for all possible behaviours of the scheduler.

Automated verification is built on the foundation of program logics and semantics. In the following we will relate a brief history of work on program logics and semantics aimed at tackling the problem of reasoning about concurrency.

1.2.1 Logics for Concurrency

Shortly after Floyd's seminal paper on reasoning about sequential programs [Floyd, 1967], Ashcroft and Manna introduced the first logic designed for reasoning about multi-threaded programs [Ashcroft and Manna, 1970]. Ashcroft and Manna's logic adapts Floyd's logic to multi-threaded systems by reasoning about the combined behaviour of all threads as a sequential program. Ashcroft and Manna's logic is appealing from the perspective of automated reasoning because techniques developed for sequential programs carry over directly to the multi-threaded case. The price paid is that the size of the sequential program that represents the interleaved behaviour of N thread is exponential in N. To get an idea for the cosmic proportions of complexity blow-up due to concurrency, consider a fairly modest-sized multi-threaded program: say that there are 1000 lines of code, executed simultaneously by 10 threads. Calculation ($1000^{10} = 10^{30}$) reveals that there are more control states in this program than there are stars in the universe (at the European Space Agency's estimation of 10^{24} stars [ESA, 2014]).

The next major development in program logics for concurrency was the discovery of thread-modular logics [Owicki and Gries, 1976, Jones, 1981]. Rather than reason about the combined behaviour of all threads as a sequential program, thread-modular logics reason about each thread individually as if it were executing in isolation. Developing a proof in a thread-modular logic can be seen as a two-step process. The first step is to derive a correctness proof for each thread (in the style of Hoare logic [Hoare, 1969]). The second step is to show that the proofs are robust in the sense that the execution of one thread does not violate the assertions in the correctness proof of another. This type of reasoning works well for programs in which threads have limited interaction, but becomes burdensome if there are sophisticated communication patterns between threads. Coping with such communication patterns

requires introducing auxiliary variables that store historical information about the program's execution to maintain the fiction that threads do not interfere with each other. Human provers are able to make elegant use of this idea, but it is poorly suited for machine provers: introducing auxiliary variables requires creativity that machines do not posses.

Techniques for automated reasoning about multi-threaded programs can trace their roots to one of the two logics above. Thus, existing techniques for reasoning about multi-threaded programs all have one thing in common: they reason about multi-threaded programs by somehow reasoning about sequential behaviour.

1.2.2 Semantics of Concurrency

A great deal of research has gone into developing a precise formal understanding of the behaviour of concurrent systems. The topic addressed by concurrency semantics defining exactly what *behaviour* means. Semantic models are necessary for developing the meta-theory of logics for reasoning about concurrent programs (when is a logic sound or complete?). In some cases, semantic models of concurrent systems are even useful as computational objects that can be analyzed algorithmically [Clarke et al., 1983]. A number of different models of concurrency have been developed. The most fundamental divide among these models is whether the behaviour of a concurrent system is *interleaved* or *truly concurrent*.

In an *interleaving* model, a behaviour of a concurrent system is identified with a sequence that interleaves the behaviour of all threads [Hoare, 1978]. This is a simple and convenient model, and this dissertation will appeal to the interleaving model throughout to make correctness arguments. However, there is a sense in which the interleaving model is over-specified. For example, consider two interleaved behaviours `x := 0; y := 0` and `y := 0; x := 0`. These behaviours are indistinguishable (in the sense that a program the executes either ends in the same state), and yet are distinct in the interleaving model. In the context of automated reasoning, this problem is manifested in the fact that there is a large number of interleaved behaviours of a concurrent system, all of which must be verified in order to verify a property of interest.

The fact that the interleaving model allows one to distinguish between equivalent behaviours suggests that the interleaving model is an approximation of the "true" behaviour of a concurrent system. In an interleaved execution, one may distinguish between *accidental* ordering of events (the scheduler executed a before b) and *causal* ordering (the execution of b *depends* on a having been executed previously). Models of *true concurrency* represent only the latter. Besides the philosophical appeal, true concurrency has practical implications for algorithmic verification, because true concurrency models have *fewer* behaviours. A simple model of true concurrency is Mazurkiewicz's trace model [Mazurkiewicz, 1986]. The idea behind this model is that independent events (such as `x := 0` and `y := 0` above) commute in any interleaving. An equivalence relation between interleavings can be defined so that two interleavings are equivalent if they differ only in the ordering of commuting events. A trace is an equivalence class of interleavings under this equivalence relation.

Interleaved traces and Mazurkiewicz traces are both *linear time*, *behavioural* models: the model defines executions of a system over one timeline. *Branching time* models define executions over possible timelines. Like linear-time models, branching time models come in both interleaving (computation trees [Clarke and Emerson, 1982]) and truly concurrent (event structures and Petri net unfoldings [Nielsen et al., 1981]) variants. In the scope of this dissertation, we will be interested only in non-reachability properties, which makes (simpler) linear-time models more appropriate. A *system-level*

model (as opposed to a *behaviour-level* model) defines a concurrent system, rather than one of its executions. Control flow graphs are the canonical interleaved system-level model, and truly concurrent systems include Petri nets [Petri, 1962] and process networks [Kahn, 1974].

1.3 My thesis

This dissertation demonstrates the thesis that tractable and precise algorithmic verification techniques can be built on a foundation that **explicitly represents parallelism**. We propose truly concurrent *logics* for reasoning about concurrent systems. More concretely,

- *Data flow graphs* can be used as a foundation for static analysis for multi-threaded programs. Data flow graphs were originally developed to expose instruction-level parallelism in sequential programs for the purpose of automatic parallelization (i.e., transforming the program to introduce parallelism to speed up computations). In Part II of this dissertation, data flow graphs are re-purposed to generate invariants for multi-threaded programs. The key idea is that the same structure that exposes parallelism in sequential programs can be used to retain parallelism in multi-threaded programs, and therefore act as a succinct and precise program representation.

  The claims of tractability and precision are demonstrated experimentally: an implementation of the techniques described in Part II is shown to have low memory and time requirements and an acceptably low (less than 30%) false alarm rate on a suite of device drivers.

- *Trace-theoretic* verification techniques can be used as a foundation software model checking for multi-threaded programs. The essential idea is to treat the verification problem as a language learning problem: the program consists of a set of traces, and the goal is to learn a language that contains all the program traces, but no traces that violate a given property. The learning approach separates the problem into two sub-problems: *constructing* a proof from examples, and *checking* that the proof is complete. Since the example traces are sequential, proof construction may re-use sequential verification techniques (e.g., Craig interpolation [McMillan, 2006]). Proof checking is an automata-theoretic problem that may re-use techniques from finite-state model checking (e.g., partial order reduction [Peled, 1993]).

  The claim of tractability is demonstrated analytically via a succinctness theorem (Section 4.4) that shows that inductive data flow graphs are small in a technical sense. We demonstrate precision by proving relative completeness results that establish the expressivity of the proof systems (Sections 4.4, 5.4).

1.4 Organization of this Dissertation

This dissertation is divided into four parts. Part I introduces the definitions and notation that will be used in the remainder of the manuscript. Parts II and III comprise the technical core of the dissertation. Part II presents techniques for static analysis of multi-threaded programs.

- Chapter 3 presents a method for generating invariants for multi-threaded programs based on *data flow graphs*. The material in this chapter is based on [Farzan and Kincaid, 2012].

Part III presents techniques for *software model checking* of multi-threaded programs. It is divided into two chapters:

- Chapter 4 presents *inductive data flow graphs*, a proof system that shares some of the key insights of data flow graphs. While data flow graphs represents parallelism in programs, *inductive* data flow graphs represent parallelism in proofs. The material in this chapter is based on [Farzan et al., 2013].
- Chapter 5 presents *proof spaces*, which may be thought of as a generalization of inductive data flow graphs to handle programs with infinitely many threads. The material in this chapter is based on [Farzan et al., 2015].

Finally, Part IV concludes with a discussion of the contributions of the dissertation (Chapter 6) and work that remains to be done (Chapter 7).

Chapter 2

Background

This chapter provides the prerequisite definitions required to put what follows on rigorous footing. Most of the terminology and notation is standard, but is defined for reference in Section 2.1. Section 2.2 defines PLIP (Parallel Linear Integer Programs), a model for parallel programs with shared memory that will be used throughout.

2.1 Preliminaries & Notation

Our notation is summarized in Figure 2.1. The symbol $\triangleq$ denotes definitional equality ("is equal to by definition"). We use λ-notation to define anonymous functions (e.g., $\lambda x.x+1$ denotes the successor function which maps each integer x to $x+1$). $\mathbb{Z}$ denotes the set of integers. $\mathbb{B}$ denotes the set of Boolean values $\{\mathit{true}, \mathit{false}\}$. Given a set A, $|A|$ denotes the cardinality of A, 2^A denotes the powerset of A, and A^* denotes the set of sequences over A, including the empty sequence which is denoted by ϵ. For a sequence π, we use $|\pi|$ to denote the length of π. For two sequences π and π', we use their juxtaposition $\pi\pi'$ to denote the concatenation of π and π'. For sets of sequences $L, L' \subseteq A^*$ ("*languages*") over a set A, we use $L \cdot L' \triangleq \{\pi\pi' : \pi \in L \wedge \pi' \in L'\}$ to denote their point-wise concatenation.

Given a map $f : X \to Y$ and elements $x \in X$ and $y \in Y$, $f[x \leftarrow y]$ denotes the function which maps x to y and every $z \in X \setminus \{x\}$ to $f(z)$:

$$f[x \leftarrow y] \triangleq \lambda z.\text{if } x = z \text{ then } y \text{ else } f(z) \ .$$

First-order logic is used throughout this dissertation. For simplicity, we will fix linear integer arithmetic as a base language for describing both programs and their properties. The syntax of linear integer arithmetic terms and formulas is given in Figure 2.2. These definitions are parameterized by a syntactic category of *variables* (V), which allows us to re-use them for several purposes. The semantics of linear integer arithmetic terms is given by an evaluation function $\mathcal{T}[\![t]\!](M)$, which maps a term $t \in \mathsf{Term}(V)$ (for some set of variables V) and a valuation $M : V \to \mathbb{Z}$ to t's interpretation in M. The semantics of formulas is given by a satisfaction relation $M \models \varphi$, which holds if the formula φ is true when interpreted over the valuation M. The semantics of linear integer arithmetic is defined formally in Figure 2.3.

We use the notation $[x \mapsto t]$ to denote a substitution that replaces the variable x with the term t and generalize the notation to parallel substitutions in the standard way (e.g., $[x_1 \mapsto t_1, x_2 \mapsto t_2]$ denotes the simultaneous substitution of t_1 for x_1 and t_2 for x_2, and $[V \mapsto V']$ denotes the substitution which maps

$\triangleq$	Definitional equality
$\mathbb{Z}$	Set of integers $\{..., -1, 0, 1, ...\}$
$\mathbb{B}$	Set of Booleans $\{true, false\}$
2^A	Power set of A
A^*	Set of finite sequences over A
$\epsilon \in A^*$	Empty sequence
$\lvert a_1 \cdots a_n \rvert \triangleq n$	Length of a sequence
$f[x \leftarrow y] \triangleq \lambda z.\text{if } x = z \text{ then } y \text{ else } f(z)$	Function update

Figure 2.1: Summary of notation.

$$\textsc{TInt}\ \frac{n \in \mathbb{Z}}{n : \mathsf{Term}(V)} \qquad \textsc{TVar}\ \frac{x \in V}{x : \mathsf{Term}(V)} \qquad \textsc{TAdd}\ \frac{t_1 : \mathsf{Term}(V) \qquad t_2 : \mathsf{Term}(V)}{t_1 + t_2 : \mathsf{Term}(V)} \qquad \textsc{TMul}\ \frac{n \in \mathbb{Z} \qquad t : \mathsf{Term}(V)}{n \times t : \mathsf{Term}(V)}$$

$$\textsc{FLt}\ \frac{t_1 : \mathsf{Term}(V) \qquad t_2 : \mathsf{Term}(V)}{t_1 < t_2 : \mathsf{Formula}(V)} \qquad \textsc{FEq}\ \frac{t_1 : \mathsf{Term}(V) \qquad t_2 : \mathsf{Term}(V)}{t_1 = t_2 : \mathsf{Formula}(V)}$$

$$\textsc{FAnd}\ \frac{\varphi : \mathsf{Formula}(V) \qquad \psi : \mathsf{Formula}(V)}{\varphi \wedge \psi : \mathsf{Formula}(V)} \qquad \textsc{FOr}\ \frac{\varphi : \mathsf{Formula}(V) \qquad \psi : \mathsf{Formula}(V)}{\varphi \vee \psi : \mathsf{Formula}(V)}$$

Figure 2.2: Syntax of linear integer arithmetic.

every variable $x \in V$ to a "primed" copy $x' \in V'$). The application of a substitution S to a formula φ (or a term t) is denoted $\varphi[S]$ (respectively $t[S]$).

2.2 Program Model

We formalize a multi-threaded imperative programming language, PLIP. This model makes a number of simplifying assumptions that will make it easier to describe the key contributions of the dissertation, without the complications involved in analyzing a fully-featured programming language.

The syntax of PLIP is defined in Figure 2.4. A PLIP program consists of a sequential program (a "*thread template*") which is executed in parallel by some set of threads. Formally, a PLIP *program* is

$M \in \mathsf{Valuation}(V) = V \to \mathbb{Z}$ Valuations

Term semantics

$$\mathcal{T}[\![\cdot]\!](\cdot) : \mathsf{Term}(V) \to \mathsf{Valuation}(V) \to \mathbb{Z}$$

$$\begin{aligned} \mathcal{T}[\![n]\!](M) &\triangleq n \\ \mathcal{T}[\![x]\!](M) &\triangleq M(x) \\ \mathcal{T}[\![t + t']\!](M) &\triangleq \mathcal{T}[\![s]\!](M) + \mathcal{T}[\![t']\!](M) \\ \mathcal{T}[\![n \times t]\!](M) &\triangleq n \times \mathcal{T}[\![t]\!](M) \end{aligned}$$

Formula semantics

$$\cdot \models \cdot : \mathsf{Valuation}(V) \to \mathsf{Formula}(V) \to \mathbb{B}$$

$$\begin{aligned} M \models t < t' &\iff \mathcal{T}[\![t]\!](M) < \mathcal{T}[\![t']\!](M) \\ M \models t = t' &\iff \mathcal{T}[\![t]\!](M) = \mathcal{T}[\![t']\!](M) \\ M \models \varphi \wedge \psi &\iff M \models \varphi \wedge M \models \psi \\ M \models \varphi \vee \psi &\iff M \models \varphi \vee M \models \psi \end{aligned}$$

Figure 2.3: Semantics of linear integer arithmetic.

defined to be a quadruple $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$. Loc is a set of *control locations*, which specifies the values that a thread's program counter may take. $\Sigma \subseteq \mathsf{Loc} \times \mathsf{Instr} \times \mathsf{Loc}$ is a set of *commands*: a command $\langle \ell, instr, \ell' \rangle$ indicates that a thread may execute the instruction *instr* to transition from ℓ to ℓ'. $\ell_{\mathsf{init}} \in \mathsf{Loc}$ is an *initial location*, which specifies where each thread begins. Finally, N is a nonempty (possibly infinite) set of *thread identifiers*, which specifies the set of threads that execute the thread template in parallel. Each PLIP instruction $instr \in \mathsf{Instr}$ (Figure 2.5) is either a deterministic assignment x `:=` t, a nondeterministic assignment x `:= *` (which assigns x an arbitrary value), a *guard* $[\varphi]$ (which does not change the program state but is blocked unless the formula φ holds), or an atomic block $\mathbf{atomic}\{instr_1; ...; instr_k\}$ consisting of a sequence of instructions that are executed as a single unit. The syntactic category Var of variables is partitioned into a category of global variables GV (which are shared by all threads) and a category LV of local variables (which are not shared).

Explicit synchronization instructions (such as locking instructions) are notably absent from this program model. Instead, threads may communicate through global memory. For example, locking can be encoded in PLIP with a global variable (say `lock`) which is 0 when the lock is free and 1 when it is held: `acquire(lock)` is encoded as $\mathbf{atomic}\{$`[lock = 0]; lock := 1`$\}$, and `release(lock)` as `lock := 0`. Explicit control flow instruction are also absent from PLIP, since control flow is encoded into the graphical structure of the thread template. Also note that threads do not have programmatic access to their identifiers (however, threads can be instrumented with code that assigns each one a unique identifier, for example beginning each thread with $\mathbf{atomic}\{$`my_id := max_id; max_id := max_id + 1`$\}$, where `my_id` is a local variable and `max_id` is a global).

Figure 2.6 gives a concrete example of a thread template for a PLIP program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ alongside a textual description (written in an unspecified imperative programming language meant to be understood informally). The set of locations Loc corresponds to the vertices of the graph, and the set of commands Σ the edges. Notice that the conditional branching in the loop is encoded as branch between two guards in the thread template, and that the assertion is encoded as a guarded branch to an error location. Supposing that the set of threads $N \triangleq \mathbb{N}$ is identified with the natural numbers, P is the program wherein there are infinitely threads, each of which executes the the code shown in the figure.

The operational semantics of PLIP programs is defined in Figure 2.7. Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ be a PLIP program. A *state* of P is a pair $\langle \rho, loc \rangle$ consisting of a store ρ which gives an interpretation to each variable and a control assignment *loc* which gives a location to each thread. Each thread $\mathtt{i} \in N$ has a distinct copy of each local variable x, which we write as $x(\mathtt{i})$ (an "indexed local variable"). We use $\mathsf{LV}(N)$ to denote the set of indexed local variables. A store $\rho : (\mathsf{GV} \cup \mathsf{LV}(N)) \to \mathbb{Z}$ is defined to be a valuation over the set of global and indexed local variables. A state $\langle \rho, loc \rangle$ is *initial* if for all $\mathtt{i} \in N$, $loc(\mathtt{i}) = \ell_{\mathsf{init}}$. We define $loc_{\mathsf{init}} : N \to \mathsf{Loc}$ to be the function such that $loc_{\mathsf{init}}(\mathtt{i}) = \ell_{\mathsf{init}}$ for all $\mathtt{i} \in N$ (so that a state $\langle \rho, loc \rangle$ is initial if and only if $loc = loc_{\mathsf{init}}$). We write $\langle \rho, loc \rangle \xrightarrow{\langle \sigma : \mathtt{i} \rangle} \langle \rho', loc' \rangle$ to denote that the program may transition from the state $\langle \rho, loc \rangle$ to the state $\langle \rho', loc' \rangle$ by thread $\mathtt{i}$ executing the command σ.

A *trace* $\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle \in \Sigma(N)^*$ of P is sequence of indexed commands. We write $s \xrightarrow{\tau} s'$ if there exists a sequence of states $s_0, ..., s_n$ such that

$$s = s_0 \xrightarrow{\langle \sigma_1 : \mathtt{i}_1 \rangle} s_1 \cdots s_{n-1} \xrightarrow{\langle \sigma_n : \mathtt{i}_n \rangle} s_n = s' \ .$$

A trace τ is *feasible* if there exists states $s_0, s \in \mathsf{State}$ such that s_0 is initial and $s_0 \xrightarrow{\tau} s$. Otherwise, τ is

$P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$	PLIP Program
$\ell \in \mathsf{Loc}$	Control locations of P
$\sigma \in \Sigma \subseteq \mathsf{Loc} \times \mathsf{Instr} \times \mathsf{Loc}$	Commands of P
$\ell_{\mathsf{init}} \in \mathsf{Loc}$	Initial location of P
$\mathtt{i} \in N$	Thread identifiers of P
GV	Global variables
LV	Local variables
$x \in \mathsf{Var} \triangleq \mathsf{GV} \cup \mathsf{LV}$	Variables

Figure 2.4: Syntax of PLIP.

infeasible. For example, the sequence

$$\langle(\ell_{\mathsf{init}}, \texttt{tmp := count}, \ell_1) : 1\rangle\langle(\ell_1, \texttt{[tmp > 0]}, \ell_2) : 1\rangle\langle(\ell_2, \texttt{count := tmp - 1}, \ell_3) : 1\rangle\langle(\ell_{\mathsf{init}}, \texttt{tmp := count}, \ell_1) : 2\rangle$$

is a feasible trace of the program in Figure 2.6 involving two threads 1 and 2. On the other hand,

$$\langle(\ell_{\mathsf{init}}, \texttt{tmp := count}, \ell_1) : 1\rangle\langle(\ell_1, \texttt{[tmp > 0]}, \ell_2) : 1\rangle\langle(\ell_2, \texttt{count := tmp - 1}, \ell_3) : 1\rangle\langle(\ell_{\mathsf{init}}, \texttt{tmp := count}, \ell_1) : 1\rangle$$

is an infeasible trace (the last command cannot be executed, because thread 1 is at ℓ_3), as is

$$\langle(\ell_{\mathsf{init}}, \texttt{tmp := count}, \ell_1) : 1\rangle\langle(\ell_1, \texttt{[tmp > 0]}, \ell_2) : 1\rangle\langle(\ell_2, \texttt{count := tmp - 1}, \ell_3) : 1\rangle\langle(\ell_{\mathsf{init}}, \texttt{[count < 0]}, \ell_{\mathsf{err}}) : 1\rangle$$

(because the guard of the last command cannot be satisfied). For brevity, when we write traces in the following we will typically omit the explicit source and destination control locations from commands. For any command $\sigma = (\ell, \mathit{instr}, \ell')$ we use $\mathsf{src}(\sigma)$ to denote σ's source location ℓ and $\mathsf{tgt}(\sigma)$ to denote σ's target location ℓ'. Given a thread $\mathtt{i}$, we extend the target function to a trace τ by defining

$$\mathsf{tgt}(\epsilon, \mathtt{i}) = \ell_{\mathsf{init}}$$

$$\mathsf{tgt}(\tau\langle\sigma : \mathtt{j}\rangle, \mathtt{i}) = \begin{cases} \mathsf{tgt}(\sigma) & \text{if } \mathtt{i} = \mathtt{j} \\ \mathsf{tgt}(\tau, \mathtt{i}) & \text{otherwise} \end{cases}$$

Supposing that thread $\mathtt{i}$ begins at the initial location ℓ_{init}, then $\mathsf{tgt}(\tau, \mathtt{i})$ is the location that thread $\mathtt{i}$ after executing τ. The set of *program traces* $\mathcal{L}(P)$ of P are the set of traces which form interleaved paths in the control flow graph. Formally, $\mathcal{L}(P)$ is the smallest set which satisfies the following: (1) $\epsilon \in \mathcal{L}(P)$, (2), if $\tau \in \mathcal{L}(P)$ and $\mathsf{tgt}(\tau, \mathtt{i}) = \mathsf{src}(\sigma)$, then $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}(P)$. Every feasible trace is a program trace, but not every program trace is necessarily feasible.

A state s is *reachable* if there is an initial state s_0 and a trace τ such that $s_0 \xrightarrow{\tau} s$. A formula φ is an *invariant* if it is satisfied by all reachable states.

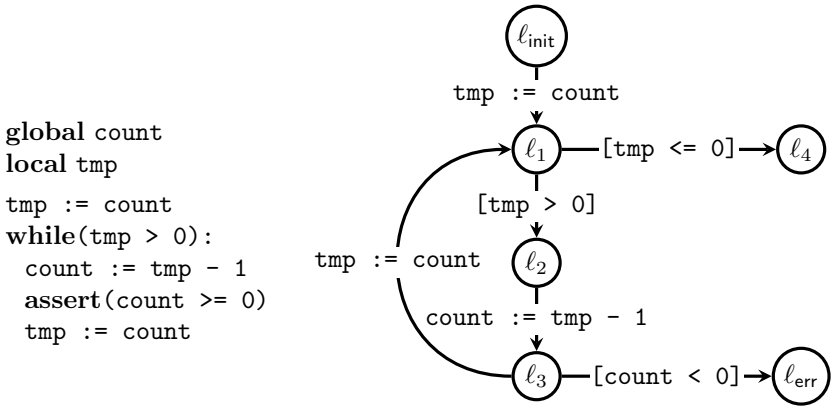
Thread stores Thread $\mathtt{i}$ can read and write its own local variables and all global variables, but the locals of other threads are hidden. It will sometimes be useful to refer to the portion of a store ρ which is accessible from a distinguished thread $\mathtt{i}$, called $\mathtt{i}$'s *thread store*. Formally, this is captured by the definition

$$\rho^{[\mathtt{i}]} \triangleq \lambda x : \mathsf{Var}.\text{if } x \in \mathsf{GV} \text{ then } \rho(x) \text{ else } \rho(x(\mathtt{i}))$$

$$\textsc{Assign}\ \frac{x \in \mathsf{Var} \qquad t : \mathsf{Term}(\mathsf{Var})}{x\ \texttt{:=}\ t : \mathsf{Instr}} \qquad \textsc{Havoc}\ \frac{x \in \mathsf{Var}}{x\ \texttt{:=}\ \texttt{*} : \mathsf{Instr}} \qquad \textsc{Assume}\ \frac{\varphi : \mathsf{Formula}(\mathsf{Var})}{[\varphi] : \mathsf{Instr}}$$

$$\textsc{Atomic}\ \frac{\mathit{instr}_1 : \mathsf{Instr} \qquad \cdots \qquad \mathit{instr}_k : \mathsf{Instr}}{\textbf{atomic}\{\mathit{instr}_1; ...; \mathit{instr}_k\} : \mathsf{Instr}}$$

Figure 2.5: Syntax of PLIP instructions.



(a) Program text. (b) Thread template.

Figure 2.6: A sample PLIP program.

which is a valuation $\rho^{[\mathtt{i}]} : \mathsf{Var} \to \mathbb{Z}$ that interprets each global variable as in ρ, and each local variable as thread `i`'s copy.

Hoare triples A *Hoare triple* $\{\varphi\}\ \tau\ \{\psi\}$ consists of a *precondition* $\varphi \in \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}(N))$, a trace $\tau \in \Sigma(N)^*$, and a *post-condition* $\psi \in \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}(N))$. We say that a Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$ is *valid* if for all program states $\langle \rho, \mathit{loc}\rangle, \langle \rho', \mathit{loc}'\rangle \in \mathsf{State}$ such that $\langle \rho, \mathit{loc}\rangle \xrightarrow{\tau} \langle \rho', \mathit{loc}'\rangle$ and $\rho \models \varphi$, we have $\rho' \models \psi$.

Transition formulas It will sometimes be useful to represent the action of an instruction by a *transition formula* $\varphi \in \mathsf{Formula}(\mathsf{Var} \cup \mathsf{Var}')$ over the set of global and local variables and their primed copies (representing the values of the variables in the pre- and post-state of the transition, respectively). The

Semantic domains

$\mathsf{LV}(N) \triangleq \{x(\mathtt{i}) : x \in \mathsf{LV}, \mathtt{i} \in N\}$	Indexed local variables
$\rho \in \mathsf{Store} \triangleq (\mathsf{GV} \cup \mathsf{LV}(N)) \to \mathbb{Z}$	Stores
$loc \in \mathsf{ControlState} \triangleq \mathsf{Loc}^N$	Control states
$s = \langle \rho, loc \rangle \in \mathsf{State} \triangleq \mathsf{Store} \times \mathsf{ControlState}$	States
$\langle \sigma : \mathtt{i} \rangle \in \Sigma(N) \triangleq \Sigma \times N$	Indexed commands
$\tau \in \Sigma(N)^*$	Traces
$ts \in \mathsf{ThreadStore} \triangleq \mathsf{Valuation}(\mathsf{GV} \cup \mathsf{LV})$	Thread store
$\rho^{[\mathtt{i}]} \triangleq \lambda x : \mathsf{GV} \cup \mathsf{LV}.\text{if } x \in \mathsf{GV} \text{ then } \rho(x) \text{ else } \rho(x(\mathtt{i}))$	

Instruction semantics

$$\mathcal{I}[\![[\varphi]]\!]_{\mathtt{i}} \triangleq \{\langle \rho, \rho \rangle : \rho^{[\mathtt{i}]} \models \varphi\}$$

$$\mathcal{I}[\![x \ \mathtt{:=}\ t]\!]_{\mathtt{i}} \triangleq \begin{cases} \{\langle \rho, \rho[x \leftarrow \mathcal{T}[\![t]\!](\rho^{[\mathtt{i}]})]\rangle\} & \text{if } x \in \mathsf{GV} \\ \{\langle \rho, \rho[x(\mathtt{i}) \leftarrow \mathcal{T}[\![t]\!](\rho^{[\mathtt{i}]})]\rangle\} & \text{if } x \in \mathsf{LV} \end{cases}$$

$$\mathcal{I}[\![x \ \mathtt{:=}\ \mathtt{*}]\!]_{\mathtt{i}} \triangleq \begin{cases} \{\langle \rho, \rho[x \leftarrow n]\rangle : n \in \mathbb{Z}\} & \text{if } x \in \mathsf{GV} \\ \{\langle \rho, \rho[x(\mathtt{i}) \leftarrow n]\rangle : n \in \mathbb{Z}\} & \text{if } x \in \mathsf{LV} \end{cases}$$

$$\mathcal{I}[\![\mathbf{atomic}\{instr_1; ...; instr_k\}]\!]_{\mathtt{i}} \triangleq \{\langle \rho_0, \rho_k \rangle : \langle \rho_0, \rho_1 \rangle \in \mathcal{I}[\![instr_1]\!]_{\mathtt{i}}, ..., \langle \rho_{k-1}, \rho_k \rangle \in \mathcal{I}[\![instr_k]\!]_{\mathtt{i}}\}$$

Labelled transition relation

$$\frac{\sigma = \langle \ell, instr, \ell' \rangle \qquad loc(\mathtt{i}) = \ell \qquad \langle \rho, \rho' \rangle \in \mathcal{I}[\![instr]\!]_{\mathtt{i}}}{\langle \rho, loc \rangle \xrightarrow{\langle \sigma : i \rangle} \langle \rho', loc[\mathtt{i} \leftarrow \ell'] \rangle}$$

$$\frac{\langle \rho_1, loc_1 \rangle \xrightarrow{\tau} \langle \rho_2, loc_2 \rangle \qquad \langle \rho_2, loc_2 \rangle \xrightarrow{\tau'} \langle \rho_3, loc_3 \rangle}{\langle \rho_1, loc_1 \rangle \xrightarrow{\tau\tau'} \langle \rho_3, loc_3 \rangle}$$

Figure 2.7: Semantics of a PLIP program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$.

transition formula representation of instructions is as follows:

$$\mathsf{Formula}[\![[\varphi]]\!] \triangleq \varphi \wedge \bigwedge_{x \in \mathsf{Var}} x' = x$$

$$\mathsf{Formula}[\![x \texttt{ := } t]\!] \triangleq x' = t \wedge \bigwedge_{y \in \mathsf{Var} \setminus \{x\}} y' = y$$

$$\mathsf{Formula}[\![x \texttt{ := *}]\!] \triangleq \bigwedge_{y \in \mathsf{Var} \setminus \{x\}} y' = y$$

$$\mathsf{Formula}[\![\mathbf{atomic}\{instr_1; ...; instr_k\}]\!] \triangleq \mathsf{Formula}[\![instr_1]\!]; \cdots; \mathsf{Formula}[\![instr_k]\!]$$

where $\varphi; \psi \triangleq \exists \mathsf{Var}''.\varphi[\mathsf{Var}' \mapsto \mathsf{Var}''] \wedge \psi[\mathsf{Var} \mapsto \mathsf{Var}'']$. Transition formulas are related to instruction semantics by the following equation, which holds for all instructions $instr : \mathsf{Instr}$ and all threads $\mathtt{i} \in N$:

$$\{\langle \rho^{[\mathtt{i}]}, \rho'^{[\mathtt{i}]} \rangle : \langle \rho, \rho' \rangle \in \mathcal{I}[\![instr]\!]_{\mathtt{i}}\} = \{\langle \underline{\rho}, \underline{\rho}' \rangle \in \mathsf{ThreadStore} \times \mathsf{ThreadStore} : [\underline{\rho}, \underline{\rho}'] \models \mathsf{Formula}[\![instr]\!]\}$$

where $[\underline{\rho}, \underline{\rho}']$ is a valuation which interprets each $x \in \mathsf{Var}$ as $\underline{\rho}(x)$ and each $x' \in \mathsf{Var}'$ as $\underline{\rho}'(x)$. That is, $\mathsf{Formula}[\![instr]\!]$ captures the effect of thread $\mathtt{i}$ executing $instr$ on the thread store of $\mathtt{i}$.

Part II

Static Analysis

Chapter 3

Data Flow Graphs

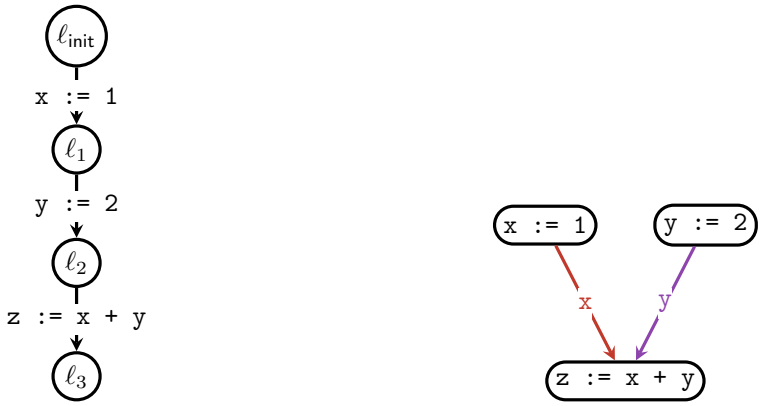
This chapter describes a method for generating invariants of concurrent programs in which an unbounded number of threads execute simultaneously. The central idea is to represent the program as a *data flow graph*. Data flow graphs explicitly represent the parallelism in a program, allowing for a succinct program representation which enables precise invariant generation.

3.1 Overview

Data flow graphs (DFGs) are a program representation that explicitly represents the *flow of data* in a program. It is instructive to contrast data flow graphs with *control flow graphs*, the dominant program representation used in program analysis (and the formalism used to describe PLIP programs in Section 2.2). Figure 3.1 gives two graphical depictions of a simple computation – one as a control flow graph, and one as a data flow graph.

The control flow graph presents a sequential view of the computation. Events are arranged in a linear order: first `x := 1` is executed, then `y := 2`, and finally `z := x + y`. Each edge represents the evolution of the computation in one time step.

The data flow graph presents a *parallel* view of the computation. Events are arranged by causality



(a) Control flow graph

(b) Data flow graph

Figure 3.1: A computation represented by a control flow graph and a data flow graphs

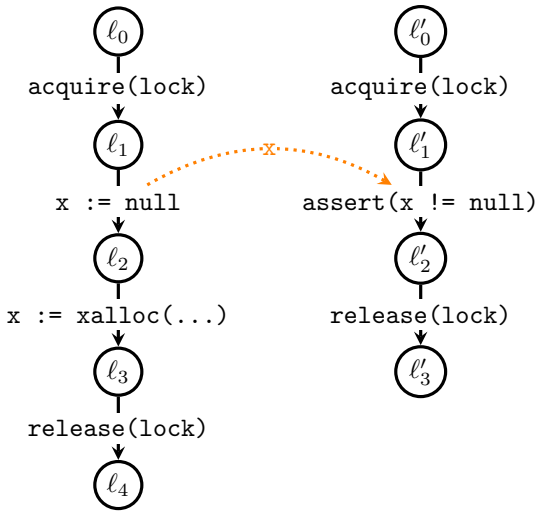


Figure 3.2: The effect of synchronization on data flow

in a *partial* order: when `z := x + y` executed, it gets the value of `x` from `x := 1` and the value of `y` from `y := 2`. These events must happen before `z := x + y`, but there is no implied order between `x := 1` and `y := 2`. The order of these two events is irrelevant to the computation, and they may even be executed in *parallel*. Each edge in the DFG represents a matching read/write relationship (some variable is written at the source of the edge and read at the target). Bifurcation in the data flow graph represents the parallelism of the computation.

Data flow graphs originated as a model of parallel processes [Kahn, 1974]. DFGs (or more accurately a variation of DFGs commonly called *dependence graphs*) gained prominence in the 1980's when they were employed by parallelizing compilers to expose latent parallelism in sequential programs [Kuck et al., 1981, Ferrante et al., 1987]. This chapter re-purposes DFGs: rather than *expose* parallelism in sequential programs for the purpose of generating efficient code, we *retain* parallelism in parallel programs for the purpose of generating invariants.

The motivation for using data flow graphs for invariant generation is that data flow graphs are both *succinct* and *precise*. DFGs achieve succinctness by abandoning the interleaving model of concurrency, the source of the combinatorial explosion which lies the heart of the difficulty in analyzing concurrent programs (see Section 1.2). Instead, DFGs are a model of *true concurrency*. Thread structure and control information are abstracted out of the DFG, replaced instead by explicit representation of thread interference. Moreover, by abstracting away thread structure, DFGs enable generating invariants for programs with infinitely many threads.

The precision of data flow graphs stems from the fact that the data flow relationship captures crucial information about synchronization. This point can be illustrated with the program snippet in Figure 3.2. In this snippet, the lock `lock` protects access to the variable `x`. The thread on the left acquires the lock, temporarily violates some invariant of `x` (e.g., that it is non-null), restores it (by allocating memory to `x`), and then releases the lock. The thread on the right acquires `lock` and asserts that the invariant `x != null` holds. The problem of verifying that the assertion holds boils down to a data flow question: *does the write* `x := null` *reach the assertion*? The answer is no, precisely because of synchronization. Synchronization is reflected in the structure of a data flow graph by the *absence* of data flow edges that are prevented by synchronization.

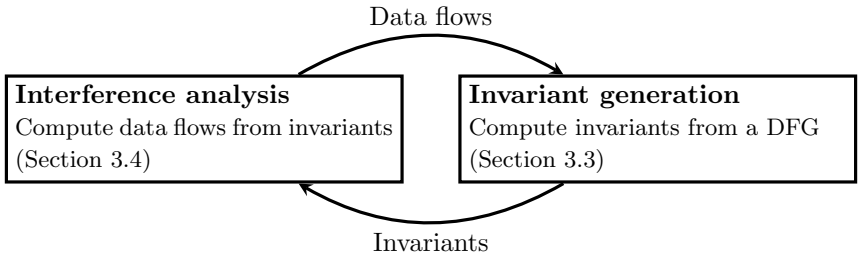


Figure 3.3: Feedback loop between interference analysis and invariant generation

The cost of using data flow graphs for invariant generation is that a data flow model must be extracted from a program's source code by some program analysis. We call this analysis an *interference analysis*, owing to the fact that its primary task is to compute inter-thread data flows. Data flow graphs have the *facility* to represent crucial information about synchronization, but utilizing this facility requires an interference analysis that accounts for synchronization. For the example in Figure 3.2, the interference analysis must be able to prove the *data* invariant that `lock` is always held at ℓ_2 to establish that the sequence of instructions

$$\langle \texttt{acquire(lock)} : 1\rangle\langle \texttt{x := null} : 1\rangle\langle \texttt{acquire(lock)} : 2\rangle\langle \texttt{assert(x != null)} : 2\rangle$$

(which would witness the data flow and thus violate the assertion) cannot be executed.

There is an apparent circularity here, depicted in Figure 3.3. In order to construct a data flow graph that reflects synchronization, we must reason about the values program variables may take at different points in the program. In order to determine the values that variables may take at different points in the program, we must construct a data flow graph. We will show that this circularity can be resolved by incorporating the interference analysis (which constructs data flow graphs using invariants) and the invariant generation procedure (which computes invariants using data flow graphs) into a feedback loop. The result is a fully automated program analysis for generating invariants for programs with infinitely many threads.

The organization of this chapter is as follows:

1. We formalize our notion of data flow graphs (Section 3.2). We define what it means for a data flow graph to represent a program.

2. We show how data flow graphs can be employed to compute invariants (Section 3.3).

3. We develop an *interference analysis* which can be used to compute a data flow graph which represents a given program (Section 3.4).

4. We give an *iterative coarsening* algorithm which incorporates interference analysis and (DFG-based) invariant generation in a feedback loop (Section 3.5).

5. We discuss extensions and issues related to the efficient implementation of the technique (Section 3.6).

6. We present Duet, a tool which implements iterative coarsening, and evaluate its performance on a suite of Linux device drivers (Section 3.7).

7. We compare the data flow graph approach to invariant generation with existing methods for generating invariants of concurrent programs (Section 3.8).

3.2 Data flow graphs

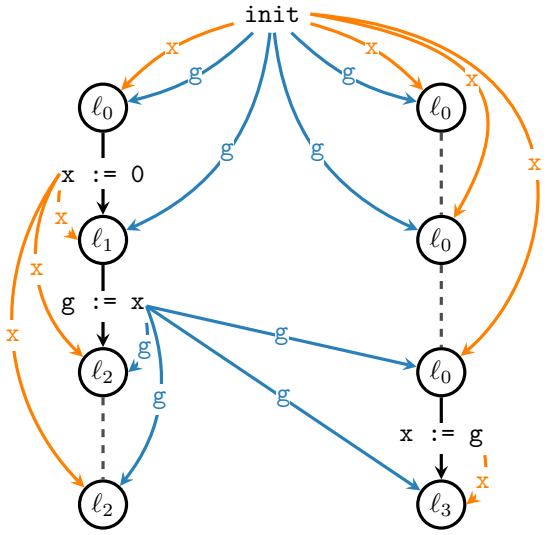
The central idea of this chapter is to represent programs as data flow graphs for the purposes of invariant generation. This section defines data flow graphs what it means to represent a program by a data flow graph.

Definition 3.2.1 (Data flow graph)**.** *Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathit{init}}, N \rangle$ be a program. A* data flow *of P is a triple $\langle u, x, \ell \rangle$ (stylized as $u \Rightarrow^x \ell$), where $u \in \Sigma_\star \triangleq \Sigma \cup \{\texttt{init}\}$ is either a command of P or a special value* `init` *(representing data flow from the initial state), $x \in \mathsf{Var}$ is a variable, and $\ell \in \mathsf{Loc}$ is a location of P. A* data flow graph *$G \subseteq \Sigma_\star \times \mathsf{Var} \times \mathsf{Loc}$ is a collection of data flows.*

We now answer the question *what is a data flow representation of a program?* The idea is intuitively simple: a data flow graph represents a program exactly when it contains every data flow $u \Rightarrow^x \ell$ that is witnessed by an execution of the program (i.e., the value of x at some occurrence of the location ℓ matches the value that was written to x at u). For example, consider the trace:

$$\langle \texttt{x := 0} : 1 \rangle \langle \texttt{g := x} : 1 \rangle \langle \texttt{x := g} : 2 \rangle$$

Suppose that the variable `x` is local and `g` is global. The data flows witnessed by this trace may be visualized as follows:



Thread 1 Thread 2

Every control point of every thread along the trace is associated with two incoming data flows, one for `x` and one for `g` (symmetric for the global variable `g` but asymmetric for the local variable `x`). The source of the data flow indicates where the value of that variable received its value. Any data flow graph that contains all the data flow edges pictured above is said to *represent* this trace. The remainder of this section formalizes condition under which a data flow graph serves as a conservative model of a program's behaviour.

First, we define a function $mod : \mathsf{Instr} \to 2^{\mathsf{Var}}$ which maps every instruction to the set of variables that it "modifies":

$$\begin{aligned} mod(x \texttt{ := } t) &\triangleq \{x\} \\ mod(x \texttt{ := *}) &\triangleq \{x\} \\ mod([\varphi]) &\triangleq \mathsf{fv}(\varphi) \\ mod(\mathbf{atomic}\{instr_1; ...; instr_k\}) &\triangleq \bigcup_{i=1}^{k} mod(instr_i) \\ mod(\ell, instr, \ell') &\triangleq mod(instr) \end{aligned}$$

This function is lifted to commands by defining $mod(\ell, instr, \ell') \triangleq mod(instr)$. A notable feature of *mod* is that a guard instruction $[\varphi]$ "modifies" every variable occurring in φ. Although $[\varphi]$ does not change the value of any variable, it changes the *information* we have about all the variables that appear in φ (e.g., after executing $[\texttt{x} = 0]$, we gain the information that $\texttt{x}$ is equal to 0). Defining *mod* in this way allows data flow graphs to take advantage of information at conditional branches.

Next, we define a function *lastmod* which maps a trace τ, a variable x, and a thread identifier $\texttt{i}$ to the last position along τ at which x was modified (thread $\texttt{i}$'s copy of x, if x is a local variable).

Definition 3.2.2. *Let $\tau = \langle\sigma_1 : \texttt{i}_1\rangle \cdots \langle\sigma_n : \texttt{i}_n\rangle$ be a trace. If x is a global variable, $lastmod(\tau, x, \texttt{j})$ is defined to be the least $k \in \{0, ..., n\}$ such that $x \notin mod(\sigma_{k'})$ for any $k' > k$. If x is a local variable, $lastmod(\tau, x, \texttt{j})$ is defined to be the least $k \in \{0, ..., n\}$ such that $x \notin mod(\sigma_{k'})$ for any $k' > k$ such that $\texttt{i}_{k'} = \texttt{j}$.*

A useful lemma is that the value of every variable at the end of an execution is equal to its value where it was last modified. Formally,

Lemma 3.2.3. *Let $\tau = \langle\sigma_1 : \texttt{i}_1\rangle \cdots \langle\sigma_n : \texttt{i}_n\rangle \in \Sigma(N)^*$ be a trace, let $\langle\rho_0, loc_{init}\rangle \xrightarrow{\langle\sigma_1:\texttt{i}_1\rangle} \cdots \xrightarrow{\langle\sigma_n:\texttt{i}_n\rangle} \langle\rho_n, loc_n\rangle$ be a corresponding execution, let $\texttt{j} \in N$ be a thread, and let $x \in \mathsf{Var}$ be a variable, and let $k = lastmod(\tau, x, \texttt{j}) \geq 1$. Then $\rho_n^{[\texttt{j}]}(x) = \rho_k^{[\texttt{i}_k]}(x)$.*

Finally, we define the set of data flows witnessed by a trace, and the representation condition that links the operational semantics of PLIP programs (Section 2.2) with data flow graphs.

Definition 3.2.4 (Witness)**.** *Let $\tau = \langle\sigma_1 : \texttt{i}_1\rangle \cdots \langle\sigma_n : \texttt{i}_n\rangle \in \Sigma(N)^*$ be a trace. A sub-trace of τ is any trace τ' such that $\tau = \tau'\tau''$ for some (possibly empty) trace τ''. We say that τ is a* witness *for a data flow $\sigma \Rightarrow^x \ell$ if there is some sub-trace τ' of τ and some thread $\texttt{i} \in N$ such that $k \triangleq lastmod(\tau', x, \texttt{i}) \geq 1$, $\sigma_k = \sigma$, and $\mathsf{tgt}(\tau', \texttt{i}) = \ell$. We say that τ is a witness for the data flow $\texttt{init} \Rightarrow^x \ell$ if there is some some sub-trace τ' of τ and some thread $\texttt{i} \in N$ such that $lastmod(\tau', x, \texttt{i}) = 0$ and $\mathsf{tgt}(\tau', \texttt{i}) = \ell$.*

Definition 3.2.5 (Representation)**.** *A DFG $G \subseteq \Sigma_\star \times \mathsf{Var} \times \mathsf{Loc}$* represents *a trace τ if G contains every data flow witnessed by τ. G represents a program P if G represents every* feasible *trace τ of P (i.e., every trace that corresponds to an execution of P).*

3.3 Generating invariants with DFGs

This section describes how a data flow graph may be used to generate invariants for a program. First we associate data flow graphs with a collecting semantics, and then a system of constrained Horn clauses. The section culminates in a theorem (Theorem 3.3.5) linking the representation condition of Definition 3.2.5 with the solutions of this constraint system.

Following the theory of abstract interpretation [Cousot and Cousot, 1977], the theoretical starting point of invariant generation is to define a *collecting semantics*. A DFG's collecting semantics specify the strongest (most precise) invariant that can be justified by the DFG. Given a program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$, the collecting semantics of a DFG of P consists of a pair of maps:

- $\mathrm{L} : \mathsf{Loc} \to 2^{\mathsf{ThreadStore}}$, which maps every location to a set of thread stores. Intuitively, L maps each location ℓ to a set of possible thread stores a thread might obtain while at ℓ.
- $\mathrm{C} : \Sigma \to 2^{\mathsf{ThreadStore}}$, which maps every command (and `init`) to a set of thread stores. Intuitively, C maps each command σ to a set of possible thread stores a thread might obtain *immediately after* executing σ).

For technical convenience, we extend C to $\Sigma_\star$ by defining $\mathrm{C}(\texttt{init}) = \mathsf{ThreadStore}$ (recall from Section 2.2 that the stores of the initial states of a program are unconstrained). Fixing a DFG G of P, its collecting semantics is defined to be the least solution to the following system of equations:

$$\begin{aligned}
\mathrm{L}(\ell) &= \{ts \in \mathsf{ThreadStore} : \forall x \in \mathsf{Var}. \exists u \in \Sigma_\star. u \Rightarrow^x \ell \in G \wedge \exists ts' \in \mathrm{C}(u). ts(x) = ts'(x)\} \\
\mathrm{C}(\texttt{init}) &= \mathsf{ThreadStore} \\
\mathrm{C}(\ell, \mathit{instr}, \ell') &= \{ts' : \exists ts \in \mathrm{L}(\ell). [ts, ts'] \models \mathsf{Formula}[\![\mathit{instr}]\!]\}
\end{aligned}$$

(Recall that $[ts, ts'] \models \mathsf{Formula}[\![\mathit{instr}]\!]$ iff the thread state of thread `i` may evolve from ts to ts' upon executing instr). The domain of the equation system, $(\mathsf{Loc} \to 2^{\mathsf{ThreadStore}}) \times (\Sigma \to 2^{\mathsf{ThreadStore}})$, is a complete lattice and the right-hand-sides of the equations are monotone, so the least fixed point is well-defined by Tarski's fixed point theorem [Tarski, 1955]. Notice that the collecting semantics is *non-relational*: for any data flow edge $u \Rightarrow^x \ell$, only the value of x flows from u to ℓ, making it impossible to correlated the values of different variables. We show how to define a relational variation of data flow graphs in Section 3.6.1.

The relationship between the collecting semantics of a DFG and the traces it represents is given by the following:

Theorem 3.3.1 (DFG Soundness)**.** *Let τ be a trace and let G be a DFG such that G represents τ. For all $\langle \rho, loc \rangle \in \mathsf{State}$ such that $\langle \rho_0, loc_{\mathsf{init}} \rangle \xrightarrow{\tau} \langle \rho, loc \rangle$ for some store ρ_0, for all $\mathtt{i} \in N$, we have that $\rho^{[\mathtt{i}]} \in L(loc(\mathtt{i}))$.*

Proof. Let G be a DFG, let $\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle$ be a trace represented by G, and let

$$\langle \rho_0, loc_{\mathsf{init}} \rangle \xrightarrow{\langle \sigma_1 : \mathtt{i}_1 \rangle} \langle \rho_1, loc_1 \rangle \cdots \langle \rho_{n-1}, loc_{n-1} \rangle \xrightarrow{\langle \sigma_n : \mathtt{i}_n \rangle} \langle \rho_n, \mathtt{i}_n \rangle$$

be an execution labeled by τ. We prove that for all $m \leq n$ and all $\mathtt{i} \in N$ that $\rho_m^{[\mathtt{i}]} \in \mathrm{L}(loc_m(\mathtt{i}))$ by induction on m.

- Base case $m = 0$: Observe that (from Definition 3.2.4 and Definition 3.2.5), if a data flow graph represents a trace, it represents all its sub-traces. Since G represents τ, we have that G represents ϵ, and thus $\texttt{init} \Rightarrow^x \ell_{\mathsf{init}} \in G$ for all $x \in \mathsf{Var}$.

$$\begin{aligned}
\mathrm{L}(\ell_{\mathsf{init}}) &\triangleq \{ts \in \mathsf{ThreadStore} : \forall x \in \mathsf{Var}.\exists u \Rightarrow^x \ell \in G.\exists ts' \in \mathrm{C}(u).ts(x) = ts'(x)\} \\
&\subseteq \{ts \in \mathsf{ThreadStore} : \forall x \in \mathsf{Var}.\exists ts' \in \mathrm{C}(\texttt{init}).ts(x) = ts'(x)\} \\
&= \{ts \in \mathsf{ThreadStore} : \forall x \in \mathsf{Var}.\exists ts' \in \mathsf{ThreadStore}.ts(x) = ts'(x)\} \\
&= \mathsf{ThreadStore}
\end{aligned}$$

  Thus, $\rho_0^{[\mathtt{i}]} \in \mathrm{L}(\ell_{\mathsf{init}})$ for all $\mathtt{i} \in N$.

- Induction step: suppose that for all $m' < m$, we have $\rho_{m'}^{[\mathtt{j}]} \in \mathrm{L}(loc_{m'}(\mathtt{j}))$ for all $\mathtt{j} \in N$. Let $\mathtt{i} \in N$. We wish to prove

$$\rho_m^{[\mathtt{i}]} \in \mathrm{L}(loc_m(\mathtt{i})) \triangleq \{ts \in \mathsf{ThreadStore} : \forall x \in \mathsf{Var}.\exists u \Rightarrow^x loc_m(\mathtt{i}) \in G.\exists ts' \in \mathrm{C}(u).ts(x) = ts'(x)\}$$

  or equivalently, that

$$\forall x \in \mathsf{Var}.\exists u \Rightarrow^x loc_m(\mathtt{i}) \in G.\exists ts' \in \mathrm{C}(u).\rho_m^{[\mathtt{i}]}(x) = ts'(x)$$

  Let $x \in \mathsf{Var}$ be a variable. Distinguish two cases:

  1. $lastmod(\tau, x, \mathtt{i}) = 0$: then $\texttt{init} \Rightarrow^x loc_m(\mathtt{i}) \in G$, and so trivially there exists $ts' \in \mathrm{C}(\texttt{init}) = \mathsf{ThreadStore}$ such that $ts'(x) = \rho_m^{[\mathtt{i}]}(x)$.
  2. $k = lastmod(\tau, x, \mathtt{i}) > 0$. Since G represents τ and τ witnesses the data flow $\sigma_k \Rightarrow^x loc_m(\mathtt{i})$, we have $\sigma_k \Rightarrow^x loc_m(\mathtt{i}) \in G$. By Lemma 3.2.3, we have that $\rho_k^{[\mathtt{i}_k]}(x) = \rho_m^{[\mathtt{i}]}(x)$, so it remains only to show that $\rho_k^{[\mathtt{i}_k]} \in \mathrm{C}(\sigma_k)$.

     By the induction hypothesis, we have $\rho_{k-1}^{[\mathtt{i}_k]} \in \mathrm{L}(loc_{k-1}(\mathtt{i}_k))$. Since $\langle \rho_{k-1}, loc_{k-1} \rangle \xrightarrow{\langle \sigma_k : \mathtt{i}_k \rangle} \langle \rho_k, loc_k \rangle$, we have $[\rho_{k-1}^{[\mathtt{i}_k]}, \rho_k^{[\mathtt{i}_k]}] \models \mathsf{Formula}[\![\sigma_k]\!]$. It then follows from the definition of C that $\rho_k^{[\mathtt{i}_k]} \in \mathrm{C}(\sigma_k)$.

□

The collecting semantics of a data flow graph provides a mathematical definition of the best invariant that can be justified by the data flow graph, but that invariant cannot be computed in general. Following recent trends in program analysis, we next provide a logical characterization of the conservative approximations of the collecting semantics. We give a system of constrained Horn clauses which can be solved using a variety of techniques, including interpretation over an abstract domain, counter-example guided abstraction refinement [Gupta et al., 2011, Grebenshchikov et al., 2012] and property-directed reachability [Hoder and Bjørner, 2012].

The class of invariants that can be generated using data flow graphs are *thread-state invariants*. Thread-state invariants are *local* in the sense that the program properties that can be described as thread-state invariants do not correlate the local variables of different threads. Formally,

Definition 3.3.2 (Thread-state invariant)**.** *A* thread-state annotation *for a program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ is a function $\Phi : \mathsf{Loc} \to \mathsf{Formula}(\mathsf{Var})$ mapping locations of P into formulas over the set local and global variables.*

We say that Φ is a thread-state invariant *if $\Phi(\ell)$ holds from the perspective of thread* `i` *in any reachable state where thread* `i` *is at ℓ. Formally, Φ is a thread-state invariant if for any reachable state $\langle \rho, loc\rangle \in \mathsf{State}$, for all thread identifiers $\mathtt{i} \in N$, we have $\rho^{[\mathtt{i}]} \models \Phi(loc(\mathtt{i}))$.*

Definition 3.3.3. *A thread-state annotation $\Phi : \mathsf{Loc} \to \mathsf{Formula}(\mathsf{Var})$ is* inductive *for a data flow graph G if for all control locations $\ell \in \mathsf{Loc}$ we have*

$$\Big[\bigwedge_{x \in \mathsf{Var}} \Big((\mathtt{init} \Rightarrow^{x} \ell \in G) \vee \bigvee_{\sigma \Rightarrow^{x} \ell \in G} \exists \mathsf{Var} \exists \mathsf{Var}' \setminus \{x'\}.\Phi(\mathsf{src}(\sigma))(\mathsf{Var}) \wedge \mathsf{Formula}[\![\sigma]\!]\Big)\Big] \Rightarrow \Phi(\ell)(\mathsf{Var}')$$

Let G be a DFG. The connection between thread-state annotations that are inductive for G and G's collecting semantics is that any inductive thread-state annotation corresponds to a solution of the semantic equations, by letting $\mathrm{L}(\ell) \triangleq \{ts \in \mathsf{ThreadStore} : ts \models \Phi(\ell)$ for all locations $\ell \in \mathsf{Loc}$ and $\mathrm{C}(\ell, instr, \ell') = \{ts' : \exists ts \in \mathrm{L}(\ell).[ts, ts'] \models \mathsf{Formula}[\![instr]\!]\}$ for all commands σ.

The following is a consequence of Theorem 3.3.1 and the fact that inductive annotations over-approximate the collecting semantics (in the above sense):

Corollary 3.3.4. *Let*

$$\langle \rho_0, loc_{\mathsf{init}}\rangle \xrightarrow{\langle \sigma_1 : \mathtt{i}_1\rangle} \langle \rho_1, loc_1\rangle \cdots \langle \rho_{n-1}, loc_{n-1}\rangle \xrightarrow{\langle \sigma_n : \mathtt{i}_n\rangle} \langle \rho_n, \mathtt{i}_n\rangle$$

be an execution, let G be a data flow graph that represents $\tau \triangleq \langle \sigma_1 : \mathtt{i}_1\rangle \cdots \langle \sigma_n : \mathtt{i}_n\rangle$ (Definition 3.2.5), and let Φ be an inductive annotation for G. Then for all threads $\mathtt{j} \in N$, $\rho^{[\mathtt{j}]} \models \Phi(loc(\mathtt{j}))$.

Theorem 3.3.5. *Let G be a data flow graph such that G represents every feasible trace τ of P, and let Φ be an inductive invariant for G. Then Φ is a thread state invariant.*

3.4 Interference analysis

The goal of interference analysis is to compute the set of all data flows that are witnessed by some *feasible* trace of the program. Membership within this set is not decidable, but we will be satisfied by a conservative approximation (i.e., we may compute more data flows than just the ones witnessed by a feasible trace). Theorem 3.3.5 guarantees that inductive invariants computed from any such conservative data flow graph yields thread-state invariants which can be used to verify program properties of interest.

The classical conservative approximation to computing a set of data flows in a sequential program is reaching definitions analysis. Reaching definitions analysis computes the set of data flows that are witnessed by any *program trace* (i.e., traces which correspond to a path in the program's control flow graph) rather than just the *feasible* traces. An analogous solution is possible in the concurrent setting but the resulting data flow graph loses crucial information about synchronization: the data flow graph contains data flows which are witnessed by traces which violate synchronization.

Consider the program in Figure 3.5. The location ℓ_{err} is unreachable, which can be inferred from an inductive annotation of the pictured data flow graph, G. But in order to conclude that an inductive

$$\textsc{init-coreach}\ \frac{}{\Phi \vdash \mathit{coreachable}(\ell_{\mathsf{init}}, \ell_{\mathsf{init}})} \qquad \textsc{coreach-sym}\ \frac{\Phi \vdash \mathit{coreachable}(\ell_1, \ell_2)}{\Phi \vdash \mathit{coreachable}(\ell_2, \ell_1)} \qquad \textsc{mayReach}\ \frac{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1, \ell_2)}{\Phi \vdash \sigma \rightsquigarrow^{x} \ell_2}$$

$$\textsc{coreach-step}\ \frac{\Phi \vdash \mathit{coreachable}(\ell_1, \ell_2) \qquad \langle \ell_1, \mathit{instr}, \ell_1' \rangle \in \Sigma \qquad \mathsf{Sat}(\Phi(\ell_1) \wedge (\exists \mathsf{LV}.\Phi(\ell_2)) \wedge \mathsf{Formula}[\![\mathit{instr}]\!])}{\mathit{coreachable}(\ell_1', \ell_2)}$$

$$\textsc{mayReach-Base}\ \frac{\Phi \vdash \mathit{coreachable}(\ell_1, \ell_2) \qquad \sigma = \langle \ell_1, \mathit{instr}, \ell_1' \rangle \in \Sigma \qquad x \in \mathit{mod}(\mathit{instr}) \qquad \mathsf{Sat}(\Phi(\ell_1) \wedge (\exists \mathsf{LV}.\Phi(\ell_2)) \wedge \mathsf{Formula}[\![\mathit{instr}]\!])}{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1', \ell_2)}$$

$$\textsc{mayReach-StepL}\ \frac{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1, \ell_2) \qquad \langle \ell_1, \mathit{instr}, \ell_1' \rangle \in \Sigma \qquad x \notin \mathit{mod}(\mathit{instr}) \qquad \mathsf{Sat}(\Phi(\ell_1) \wedge (\exists \mathsf{LV}.\Phi(\ell_2)) \wedge \mathsf{Formula}[\![\mathit{instr}]\!])}{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1', \ell_2)}$$

$$\textsc{mayReach-StepR}\ \frac{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1, \ell_2) \qquad \langle \ell_2, \mathit{instr}, \ell_2' \rangle \in \Sigma \qquad x \notin \mathit{mod}(\mathit{instr}) \qquad \mathsf{Sat}((\exists \mathsf{LV}.\Phi(\ell_1)) \wedge \Phi(\ell_2) \wedge \mathsf{Formula}[\![\mathit{instr}]\!])}{\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1, \ell_2')}$$

Figure 3.4: Interference analysis, for a program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ and thread-state annotation Φ.

annotation of G corresponds to a thread-state invariant for the program, we must show that G represents every feasible trace (Theorem 3.3.5). In particular, we must be able to verify that the data flow $(\texttt{x := 1}) \Rightarrow^{\texttt{x}} \ell_2$ (which does *not* belong to G) is *not* witnessed by a feasible trace. This is true, but it requires reasoning about synchronization on the `flag` variable. There can be only one thread with its program counter in $\{\ell_1, \ell_2, \ell_3\}$ at a time: whenever some thread is in one of those locations, we are assured that `flag` is equal to 1, and a second thread is blocked from entering ℓ_1 as long as that is the case.

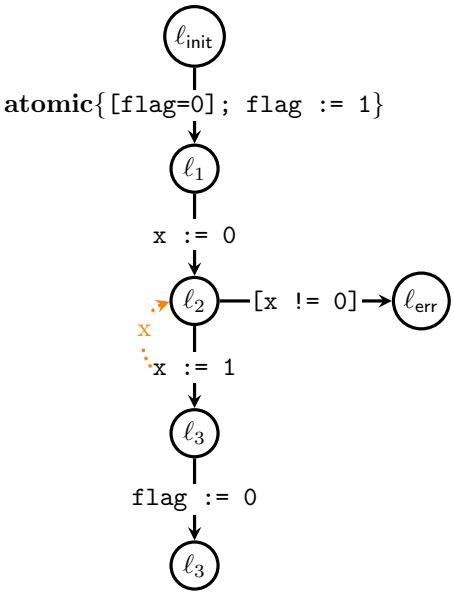
The interference analysis accommodates reasoning about data by way of a candidate thread-state annotation Φ, which is taken as an input to the analysis. (The circularity involved in computing thread-state invariants from DFGs and computing DFGs from thread-state invariants is resolved in Section 3.5.) Φ is used to approximate feasibility within the interference analysis: the analysis computes the set of data flows that are witnessed by traces which are feasible "up to Φ." Under the assumption that Φ is a thread-state *invariant*, every trace which is feasible is feasible up to Φ, and thus the interference analysis is conservative.

Suppose that a thread-state annotation Φ is given. We say that a trace τ is Φ-feasible ("feasible up to Φ") if every command along τ is enabled, in the sense that its guard is consistent with the annotation at the control point in which it is executed. Formally:

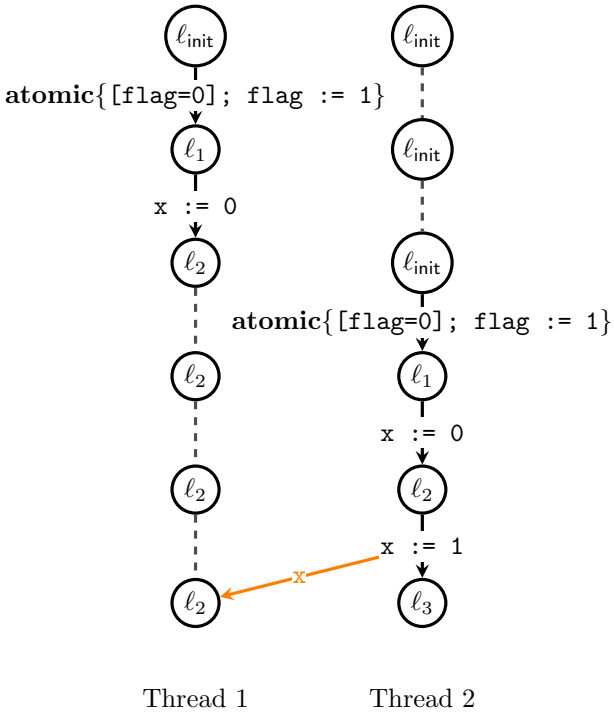
Definition 3.4.1. *Let τ be a trace and Φ be an annotation. Then τ is* Φ-feasible *if:*

- $\tau = \epsilon$, *or*
- $\tau = \tau' \langle \sigma : \mathtt{i} \rangle$, *where τ' is an Φ-feasible trace, and the following formula is satisfiable:*

$$\mathit{Enabled}(\tau', \sigma, \mathtt{i}) \triangleq \Big(\bigwedge_{\mathtt{j} \neq \mathtt{i} \in N} \exists \mathit{LV}.\Phi(\mathit{tgt}(\tau', \mathtt{j}))) \wedge \Phi(\mathit{tgt}(\tau', \mathtt{i}))) \wedge \mathit{Formula}[\![\sigma]\!]$$



(a) A thread template along with a data flow edge prevented by synchronization. `flag` and `x` are global variables, and the thread template is executed by infinitely many threads, with thread identifiers drawn from $\mathbb{N}$.



(b) An infeasible trace that witnesses the data flow

Figure 3.5: Synchronization and data flows

Example 3.4.2. *Consider the example program from Figure 3.5. Suppose that Φ is the thread-state annotation which maps every location to true. Then the trace in Figure 3.5b is Φ-feasible.*

On the other hand, suppose that Ψ is any thread-state annotation which maps ℓ_2 to `flag` $= 1$. *Then the trace in Figure 3.5b is not Ψ-feasible. Abbreviating the command*

$$(\ell_{init}, \textbf{atomic}\{\texttt{[flag=0]; flag := 1}\}, \ell_1)$$

by σ_{tas}, this trace is infeasible because it contains the following infeasible sub-trace:

$$\underbrace{\langle \sigma_{tas} : 1\rangle\langle(\ell_1, \texttt{x := 0}, \ell_2) : 1\rangle}_{\tau'}\langle \sigma_{tas} : 2\rangle \ .$$

This sub-trace is infeasible because $Enabled(\tau', \sigma_{tas}, 2)$ is inconsistent, which follows from the following calculation:

$$\begin{aligned} Enabled(\tau', \sigma_{tas}, 2) &\triangleq \Big(\bigwedge_{\texttt{j}\neq 2 \in N} \exists LV.\Psi(tgt(\tau', \texttt{j}))\Big) \wedge \Psi(tgt(\tau', 2))) \wedge Formula[\![\sigma_{tas}]\!] \\ &\models (\exists LV.\Psi(tgt(\tau', 1))) \wedge Formula[\![\sigma_{tas}]\!] \\ &= (\exists LV.\Psi(\ell_2)) \wedge Formula[\![\sigma_{tas}]\!] \\ &\equiv \texttt{flag} = 1 \wedge \texttt{flag} = 0 \wedge \texttt{flag}' = 1 \wedge \texttt{x}' = \texttt{x} \\ &\equiv false \end{aligned}$$

A key property of Φ-feasibility is that it is preserved under thread projection: if we take any Φ-feasible trace and delete the commands executed by any thread, then the resulting trace is also Φ-feasible. Intuitively, this holds because the condition for extending a Φ-feasible trace τ by an action $\langle \sigma : \mathtt{i} \rangle$ (that is, the satisfiability of the formula $\mathit{Enabled}(\tau, \sigma, \mathtt{i})$) depends only on the annotations at the end locations of each thread. The significance of the projection result is that all data flows that are witnessed by a Φ-feasible trace are witnessed by a Φ-feasible trace involving at most two threads. As a result, our interference analysis may reason about the program two threads at a time, regardless of how many threads may execute in parallel.

Towards formalizing the projection property, for any trace τ and any subset of threads $M \subseteq N$, we define $\tau|_M$ to be the result of deleting the commands executed by any thread *not* in M:

$$\epsilon|_M \triangleq \epsilon$$

$$\tau\langle \sigma : \mathtt{i} \rangle|_M \triangleq \begin{cases} \tau|_M \langle \sigma : \mathtt{i} \rangle & \text{if } \mathtt{i} \in M \\ \tau|_M & \text{if } \mathtt{i} \notin M \end{cases}$$

Lemma 3.4.3 (Projection)**.** *Let Φ be an annotation such that $\Phi(\ell) = \mathit{true}$, and let $M \subseteq N$ be a set of threads. For any trace τ, if τ is Φ-feasible, then $\tau|_M$ is Φ-feasible.*

Proof. We proceed by induction on τ. The base case is obvious. For the inductive step, let $\tau\langle \sigma : \mathtt{i} \rangle$ be a Φ-feasible trace, and assume that $\tau|_M$ is Φ-feasible. If $\mathtt{i} \notin M$, then $\tau\langle \sigma : \mathtt{i} \rangle|_M = \tau|_M$, and the result is immediate from the induction hypothesis.

If $\mathtt{i} \in M$, then $\tau\langle \sigma : \mathtt{i} \rangle|_M = \tau|_M \langle \sigma : \mathtt{i} \rangle$. By the induction hypothesis, $\tau|_M$ is Φ-feasible, so we need only to show that the formula

$$\Big(\bigwedge_{\mathtt{j} \neq \mathtt{i} \in N} \exists \mathsf{LV}.\Phi(\mathsf{tgt}(\tau|_M, \mathtt{j}))) \wedge \Phi(\mathsf{tgt}(\tau|_M, \mathtt{i}))) \wedge \mathsf{Formula}[\![\sigma]\!]$$

is satisfiable. Since $\tau\langle \sigma : \mathtt{i} \rangle$ is Φ-feasible, we have that the formula

$$\Big(\bigwedge_{\mathtt{j} \neq \mathtt{i} \in N} \exists \mathsf{LV}.\Phi(\mathsf{tgt}(\tau, \mathtt{j}))) \wedge \Phi(\mathsf{tgt}(\tau, \mathtt{i}))) \wedge \mathsf{Formula}[\![\sigma]\!]$$

is satisfiable. Thus, it is sufficient to show that for all threads $\mathtt{k} \in N$, $\Phi(\mathsf{tgt}(\tau, \mathtt{k})) \models \Phi(\mathsf{tgt}(\tau|_M, \mathtt{k}))$. Let $\mathtt{k} \in N$, and distinguish two cases:

1. $\mathtt{k} \in M$: then $\mathsf{tgt}(\tau|_M, \mathtt{k}) = \mathsf{tgt}(\tau, \mathtt{k})$, so trivially $\Phi(\mathsf{tgt}(\tau, \mathtt{k})) \models \Phi(\mathsf{tgt}(\tau|_M, \mathtt{k}))$.
2. $\mathtt{k} \notin M$: then $\mathsf{tgt}(\tau|_M, \mathtt{k}) = \ell_{\mathsf{init}}$. Since $\Phi(\ell_{\mathsf{init}}) = \mathit{true}$ by the assumption of the lemma, we have $\Phi(\mathsf{tgt}(\tau, \mathtt{k})) \models \mathit{true} = \Phi(\mathsf{tgt}(\tau|_M, \mathtt{k}))$. □

Suppose that Φ is a thread-state annotation (such that $\Phi(\ell_{\mathsf{init}}) = \mathit{true}$). The projection lemma shows that any data flow $u \Rightarrow^x \ell$ that is witnessed by a Φ-feasible trace is also witnessed by a Φ-feasible trace involving only one or two threads.

3.4.1 Inferring data flow edges

The projection property of Φ-feasible traces means that to compute all *inter-thread* data flows that are witnessed by a Φ-feasible trace, it is sufficient to consider traces involving only two threads. Without

loss of generality, we refer to the two threads as Thread 1 and Thread 2. A proof calculus for Φ-feasible data flows appears in Figure 3.4, which serves as a declarative specification of our interference analysis. There are three types of judgments in the calculus:

1. $\Phi \vdash \sigma \rightsquigarrow^x \ell$: there is a Φ-feasible witness for the data flow $\sigma \Rightarrow^x \ell$.
2. $\Phi \vdash \mathit{coreachable}(\ell_1, \ell_2)$: there is a Φ-feasible trace τ such that $\ell_1 = \mathtt{tgt}(\tau, 1)$ and $\ell_2 = \mathtt{tgt}(\tau, 2)$
3. $\Phi \vdash \mathit{mayReach}(\sigma, x, \ell_1, \ell_2)$: there is some Φ-feasible trace $\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle$ such that $\sigma = \sigma_{\mathit{lastmod}(\tau,x,1)}$, $\ell_1 = \mathtt{tgt}(\tau, 1)$, and $\ell_2 = \mathtt{tgt}(\tau, 2)$.

These inference rules are sound and complete for determining whether a witness for an inter-thread data flow edge exists – intra-thread data flows can be computed independently using a standard sequential reaching definitions analysis.

Lemma 3.4.4 (Interference analysis soundness & completeness)**.** *Let $u \in \Sigma_\star$, $x \in \mathsf{Var}$, and $\ell \in \mathsf{Loc}$. Let Φ be a thread-state annotation with $\Phi(\ell_{\mathit{init}}) = \mathit{true}$. There exists a Φ-feasible trace that witnesses data flow the $u \Rightarrow^x \ell$ iff there exists a single-threaded witness, or if $\Phi \vdash u \rightsquigarrow^x \ell$ can be derived using inference rules in Figure 3.4.*

Since the set of locations and the set of variables are finite, the set of derivable judgments of the calculus is finite. We may compute all derivable judgments by iteratively applying the inference rules until no new judgments are deduced (i.e., until a fixed point is reached). We discuss practical considerations of this analysis in Section 3.6.2.

Corollary 3.4.5. *Let $u \in \Sigma_\star$, $x \in \mathsf{Var}$, and $\ell \in \mathsf{Loc}$. Let Φ be a thread-state invariant. Let G be a data flow graph consisting of all data flows $u \Rightarrow^x \ell$ such that either (1) $u \Rightarrow^x \ell$ is a sequential reaching definition or (2) $\Phi \vdash u \rightsquigarrow^x \ell$ can be derived using inference rules in Figure 3.4. Then G represents the program P (Definition 3.2.5).*

3.5 Iterative coarsening

There is a circularity between the previous two sections:

1. In Section 3.3, we showed how to compute a thread state invariant from a data flow graph (Corollary 3.3.4)
2. In Section 3.4, we showed how to compute a data flow graph from a thread-state invariant (Corollary 3.4.5)

This section resolves the circularity by incorporating both components into a feedback loop. The resulting algorithm is Coarsen, pictured in Algorithm 1. Given a program with any number of threads (even infinite), Coarsen computes a DFG that represents that program as well as an annotation that is inductive for that DFG.

This algorithm makes use of the following auxiliary functions:

- SequentialDFG(P) computes a sequential data flow graph for P. This computation is a standard sequential reaching definitions analysis [Kildall, 1973]. This graph contains all intra-thread data flow edges, including all those for local variables, and all those originating from `init`.

Input : A program $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$
Output: A thread-state invariant for P
$G \leftarrow$ SequentialDFG(P);
repeat
  $\Phi \leftarrow$ InductiveInvariant(G);
  $G' \leftarrow G$;
  $G \leftarrow G' \cup$ FeasibleDataflows(P, Φ);
until $G' = G$;
return Φ

Algorithm 1: COARSEN

- InductiveInvariants(G) computes an annotation that is inductive for the DFG G, as described in Section 3.3.
- FeasibleDataflows(P, Φ) computes the set $\{\sigma \Rightarrow^x \ell : \Phi \vdash \sigma \rightsquigarrow^x \ell\}$ of Φ-feasible inter-thread data flows, as described in Section 3.4.

Given a program P, COARSEN begins by computing a data flow graph G_1 with only intra-thread (sequential) data flow edges. It then computes an inductive annotation Φ_1 for G_1 as discussed in Section 3.2. This annotation Φ_1 is used as input to the interference analysis of Section 3.4, which computes the set of Φ_1-feasible data flow edges and adds them to G_1 to obtain a DFG G_2. After adding these edges, a new (possibly weaker) annotation is computed that is inductive for G_2. This process continues until a fixed point is reached; that is, until we reach some k such that $G_k = G_{k+1}$. At this point, G_k represents the program P and Φ_k is inductive for G_k, and therefore Φ_k over-approximates the reachable thread states of P by Theorem 3.3.1.

The correctness condition of Algorithm 1 is stated in the following theorem:

Theorem 3.5.1 (Soundness)**.** *For any program P,* COARSEN *computes an annotation Φ and a DFG G such that G represents P, and for every reachable state $\langle \rho, loc \rangle$ of P and thread* `i`*, $\rho^{[\mathtt{i}]} \models \Phi(loc(\mathtt{i}))$.*

Proof. Let Φ and G be the annotation and data flow graph computed by COARSEN. The termination condition of COARSEN implies that (I) Φ is inductive for G and that (II) every Φ-feasible trace of P is represented by G.

We first prove that every feasible trace τ of P is Φ-feasible by induction on τ.

The base case is trivial, since ϵ is Φ-feasible for any Φ. For the induction step, assume that $\tau\langle\sigma : \mathtt{i}\rangle$ is a feasible trace and τ is Φ-feasible; we must prove that $\tau\langle\sigma : \mathtt{i}\rangle$ is Φ-feasible. Since $\tau\langle\sigma : \mathtt{i}\rangle$ is feasible, there exists $\rho_0 \in \mathsf{Store}$ and $\langle \rho, loc\rangle, \langle \rho', loc'\rangle \in \mathsf{State}$ such that $\langle \rho_0, loc_{\mathsf{init}}\rangle \xrightarrow{\tau} \langle \rho, loc\rangle \xrightarrow{\langle\sigma:\mathtt{i}\rangle} \langle \rho', loc'\rangle$.

Since τ is Φ-feasible, τ is represented by G (by (II)). Since Φ is inductive for G (by (I)), we may conclude from Corollary 3.3.4 that for all threads $\mathtt{j} \in N$, $\rho^{[\mathtt{j}]} \models \Phi(loc(\mathtt{j}))$. Since $\rho^{[\mathtt{j}]}$ and $\rho^{[\mathtt{i}]}$ agree on the interpretation of global variables, it follows that for all $\mathtt{j} \in N$, $\rho^{[\mathtt{i}]} \models \exists \mathsf{LV}.\Phi(loc(\mathtt{j}))$. It follows that the structure which interprets each variable in Var as $\rho^{[\mathtt{i}]}$ and each primed variable in Var' as $\rho'^{[\mathtt{i}]}$ is a model of the formula

$$\mathit{Enabled}(\tau, \sigma, \mathtt{i}) = \Big(\bigwedge_{\mathtt{j} \neq \mathtt{i} \in N} \exists \mathsf{LV}.\Phi(\mathsf{tgt}(\tau, \mathtt{j}))\Big) \wedge \Phi(\mathsf{tgt}(\tau, \mathtt{i}))) \wedge \mathsf{Formula}[\![\sigma]\!]$$

Thus, $\mathit{Enabled}(\tau, \sigma, \mathtt{i})$ is satisfiable, and $\tau\langle\sigma : \mathtt{i}\rangle$ is Φ-feasible.

Since every feasible trace is Φ-feasible, and every Φ-feasible trace is represented by G, every feasible trace is represented by G, and so P is represented by G. From Theorem 3.3.5, we may conclude that for every reachable state $\langle \rho, loc \rangle$ of P and thread $\mathtt{i}$, $\rho^{[\mathtt{i}]} \models \Phi(loc(\mathtt{i}))$.

□

3.6 Discussion

This section discusses some extensions and algorithmic improvements that are built into Duet, our implementation Algorithm 1.

3.6.1 Relational abstract domains

In the collecting semantics for DFGs given in Section 3.3, the values of different variables cannot be correlated. That is, a DFG cannot be used to prove that two variables are equal at a given program location (unless the DFG can prove the stronger non-relational property that both variables are equal to the same constant). This suggests that DFGs are appropriate for analyses based on *non-relational* abstract domains, such as intervals, signs, or the even/odd domain, which are incapable of representing relationships between variables anyway. Intuitively, the reason for this limitation is that each edge of the data flow graph carries the value of one variable.

We now define a relational variation of DFGs which allows variables to be correlated. The idea is to label data flow edges by *sets* of variables, indicating that the value of every variable in this set flows from the source to the target. Since a value for each $x \in X$ flows along such an edge $u \Rightarrow^X v$, relationships between variables in X can be maintained.

A particularly simple instance of this idea is to create a partition $\mathbb{P}$ of the set of variables Var into semantically related sets. Intuitively, we can think of each cell $X \in \mathbb{P}$ as a record-typed variable, with one field for each $x \in X$. For a given partition $\mathbb{P}$ of Var, the collecting semantics for a relational DFG G with $\mathbb{P}$-labeled edges is given by the following:

$$\begin{aligned} \mathrm{L}(\ell) &= \{ts \in \mathsf{ThreadStore} : \forall X \in \mathbb{P}. \exists u \Rightarrow^X \ell \in G. \exists ts' \in \mathrm{C}(u). \forall x \in X. ts(x) = ts'(x)\} \\ \mathrm{C}(\mathtt{init}) &= \mathsf{ThreadStore} \\ \mathrm{C}(\ell, instr, \ell') &= \{ts' : \exists ts \in \mathrm{L}(\ell). [ts, ts'] \models \mathsf{Formula}[\![\sigma]\!]\} \end{aligned}$$

The interference analysis of Section 3.4 must then be adapted to infer relational data flow edges. Towards this end, we redefine *mod* to act on cells rather than variables as follows:

$$mod_{\mathbb{P}}(instr) = \{X \in \mathbb{P} : mod(instr) \cap X \neq \emptyset\}$$

By re-instantiating the interference analysis in Figure 3.4 with $mod_{\mathbb{P}}$ in place of *mod*, we obtain an algorithm for calculating data flows in a relational DFG.

A relational data flow graph is not necessarily more precise than a non-relational one, a phenomenon which can be observed in the experiments in Section 3.7. The benefit of relational data flow graphs is that grouping variables together makes it possible to infer relationships between variables (which may result in a more precise analysis). On the other hand, grouping variables together can create spurious data flows that do not exist in a non-relational data flow graphs.

DUET uses Algorithm 2 to partition variables into related sets. The variable packing technique employed by the Astrée program analyzer [Blanchet et al., 2003] solves a similar problem. However, their algorithm exploits the sequential structure of the program which is not as informative for concurrent programs. While simple, our algorithm is effective in our experiments on Boolean abstractions of Linux device drivers reported in Section 3.7.

Input : Program P
Output: A partition of Var
`/* ℙ is a disjoint set data structure. Initially each cell is a singleton. */`
$\mathbb{P} \leftarrow \{\{x\} : x \in \mathsf{Var}\}$;
foreach *instruction instr in* P **do**
  if $instr = x$ `:=` t **then**
    $vs \leftarrow \{x\} \cup \mathsf{fv}(t)$;
    if $|vs| = 2 \wedge (vs \subseteq LV \vee vs \subseteq GV)$ **then**
      Merge the partitions of each $x \in vs$;
    end
  else if $instr = [\varphi]$ **then**
    if $fv(\varphi) \subseteq LV \vee fv(\varphi) \subseteq GV$ **then**
      Merge the partitions of each $x \in \mathsf{fv}(\varphi)$;
    end
  else
    skip;
  end
end
return $\mathbb{P}$

Algorithm 2: Variable partitioning algorithm

3.6.2 Efficient implementation of interference analysis

DUET implements the interference analysis presented in Section 3.4 by encoding the inference rules in Figure 3.4 as a Datalog program. The set of all derivable judgments is computed by the Datalog solver bddbddb [Whaley and Lam, 2004], which efficiently represents relations using binary decision diagrams (BDDs). The remainder of this section discusses the details of the Datalog encoding.

Several of the inference rules (COREACH-STEP, MAYREACH-BASE, MAYREACH-STEPL, MAYREACH-STEPR) in Figure 3.4 rely on checking satisfiability of a formula computed from the thread state annotation Φ and the defining formula of some instruction. This check can be implemented in datalog by pre-computing a finite relation $enabled \subseteq \mathsf{Loc} \times \Sigma$, where $enabled(\ell, \sigma)$ holds if and only if the formula

$$\Phi(\mathsf{src}(\sigma)) \wedge (\exists \mathsf{LV}.\Phi(\ell)) \wedge \mathsf{Formula}[\![\sigma]\!]$$

is satisfiable. Pre-computing *enabled* requires issuing $|\mathsf{Loc}| \times |\Sigma|$ satisfiability queries.

An alternative method used by DUET is to implement a finite-state over-approximation of the satisfiability queries, which can be answered internally by the datalog solver. The finite-state approximation is defined by a (finite) set $\mathcal{C} \subseteq \mathsf{Formula}(\mathsf{GV})$ of formulas over the set of global variables, which we call *observable conditions*. The set of observable conditions and a thread-state annotation Φ define a pair of functions:

Device Drivers	#assertions	DUET: Interval Analysis		DUET: Octagon Analysis	
		safe	time	safe	time
i8xx_tco	90	75	1m51s	71	1m25s
ib700wdt	75	64	30s	64	20s
machzwd	87	73	39s	67	14m44s
mixcomwd	91	72	22s	74	25
pcwd	240	147	2m43s	145	23m48s
pcwd_pci	204	187	2m18s	188	2m59s
sbc60xxwdt	91	77	28s	69	11m27s
sc520_wdt	85	71	28s	65	13m20s
sc1200wdt	77	66	34s	66	33s
smsc37b787_wdt	93	80	47s	80	47s
w83877f_wdt	92	78	29s	72	13m24s
w83977f_wdt	101	90	34s	82	34s
wdt	99	88	25s	86	25s
wdt977	88	77	27s	75	28s
wdt_pci	84	67	33s	66	5m33s
total	1597	1312	13m9s	1277	90m21s

Table 3.1: DUET's Performance on integer programs, run on an 3.16GHz Intel® Core 2™ machine with 4GB of RAM.

- $\Phi^{\mathcal{C}} : \mathsf{Loc} \to 2^{\mathcal{C}}$ maps each location ℓ to the set of all observable conditions entailed by $\Phi(\ell)$:

$$\Phi^{\mathcal{C}}(\ell) \triangleq \{\varphi \in \mathcal{C} : \Phi(\ell) \models \varphi\}$$

- $\mathit{guard}^{\mathcal{C}} : \Sigma \to 2^{\mathcal{C}}$ maps every command $\sigma = \langle \ell, \mathit{instr}, \ell' \rangle$ to the set of observable conditions that do *not* hold in any state where σ is executable:

$$\mathit{guard}(\sigma) \triangleq \{\varphi \in \mathcal{C} : \neg\mathsf{Sat}(\varphi \wedge \Phi(\ell) \wedge \mathsf{Formula}[\![\mathit{instr}]\!])\}$$

An abstract *enabled* relation can then be defined by:

$$\mathit{enabled}^{\mathcal{C}}(\ell, \sigma) \iff \Phi^{\mathcal{C}}(\ell) \cap \Phi^{\mathcal{C}}(\mathsf{src}(\ell)) = \emptyset \ .$$

By pre-computing $\mathit{guard}^{\mathcal{C}}$ and $\Phi^{\mathcal{C}}$ rather than *enabled*, we issue $|\mathsf{Loc}| + |\Sigma|$ satisfiability queries rather than $|\mathsf{Loc}| \times |\Sigma|$.

In DUET, the set of observable conditions is taken to be all global guards appearing in the program P:

$$\mathcal{C} \triangleq \{\varphi \in \mathsf{Formula}(\mathsf{GV}) : [\varphi] \text{ is an instruction of } P\} \ .$$

3.7 Experiments

The approach presented in this chapter is implemented in a tool called DUET. We used a benchmark suite of 15 Linux device drivers to evaluate DUET. Additionally, we ran DUET on the set of Boolean programs generated by SatAbs [Clarke et al., 2005] from these Linux drivers to compare DUET with

techniques for verifying Boolean programs with infinitely many threads.

3.7.1 Implementation

Duet is written in OCaml, and makes use of the CIL front-end for the C language [Necula et al., 2002] and the goto program front-end distributed with CBMC [Clarke et al., 2004]. Duet's abstract interpreter uses the APRON library [Bertrand and Miné, 2009] for its numerical abstract domains. We use the BDD-based Datalog implementation bddbddb [Whaley and Lam, 2004] to perform the interference analysis described in Section 3.4. Duet accepts three types of inputs: (1) C programs using *pthreads* library for thread operations, (2) Boolean programs in the input language of *Boom* [Kaiser et al., 2010], or (3) goto programs, as produced by the goto-cc C/C++ front-end (part of the CProver project [Alglave et al., 2011]). Duet's analysis is intra-procedural, and inlines function calls to perform whole-program analysis.

Interval and Octagon Configurations Duet is equipped with two configurations: Interval Analysis and Octagon Analysis. The Interval Analysis configuration generates invariants from DFGs using the interval abstract domain (each invariant assertion consists of a conjunction of assertions of the form $\pm x \leq c$, where x is a program variable and c is an integer constant). The Octagon Analysis configuration uses the octagon abstract domain [Miné, 2006], a more expressive domain that allows representing relationships between variables (each invariant assertion consists of a conjunction of interval constraints and assertions of the form $\pm x \pm y \leq c$, where x and y are program variables and c is an integer constant). Since each edge of a DFG carries the values of a single variable, standard DFGs cannot make use of the extra expressive power of octagons. However, we may group variables into packs and construct a relational DFG with pack-labelled edges (as described in Section 3.6.1). This enables the octagon domain to infer relationships between variables, but also introduces spurious data flows. Thus, the two configurations of Duet differ in two ways: they use different abstract domains (interval vs. octagon) and different variations of data flow graphs (standard vs. relational). The salient question concerning these two configurations is: *does the increased precision of the octagon abstract domain outweigh the decreased precision of the relational data flow graph?*

Alias Analysis. Duet uses a type-based alias analysis to compute data flows via reads and writes through pointers. For each variable whose address is not taken, we assign a memory location that receives strong updates. For every type in the program, we assign a memory location that receives weak updates, and each access path of that type (other than variables whose address is not taken) is considered to be a reference to that memory location. The interference analysis implemented in Duet operates on these memory locations rather than variables. This scheme is sound under the assumption that pointer-typed expressions are never cast. Aliasing is not a big factor in proving array bounds and integer overflow properties in the device drivers we considered in our evaluation, so we expect the consequences of our unsound and imprecise alias analysis to be small.

3.7.2 Evaluation

Below, we provide the results of experimenting with Duet on a collection of Linux device drivers and on Boolean abstractions of those drivers. Test harnesses and kernel models are generated by DDVerify

[Witkowski et al., 2007]. The test harness creates an unbounded number of client threads that access the driver simultaneously.

Integer Programs

Table 3.1 presents the result of running Duet on a collection of 15 Linux device drivers written in C.

DDVerify and goto-cc are used to (automatically) process each driver into a fully-inlined goto program annotated with assertions that check array bounds and integer overflows/underflows. Duet's Interval Analysis configuration proves most of the assertions correct across all 15 benchmarks (1312 out of a total 1597), and does so in 13 minutes. The Octagon Analysis configuration is slightly worse, proving 1277 assertions correct in 90 minutes. Thus, this experiment demonstrates a case where the precision of the data flow graph plays a larger role than the precision of the abstract domain. A possible explanation of this behaviour is that device drivers make heavy use of fixed-size arrays, so that interval analysis is sufficient for proving array bounds properties.

Boolean Programs

Although Boolean programs are not the target of this work, we experimented with them for two reasons: (1) verification of Boolean programs with infinitely many threads is comparatively well-studied and verification tools and there are publicly-available verifiers (2) there is no aliasing present in Boolean programs, which limits the scope of implementation-related imprecision for a better evaluation of the core method.

We compared Duet against two techniques for verifying Boolean programs with infinitely many threads: dynamic cutoff detection (DCD) [Kaiser et al., 2010] and linear interfaces (LI) [La Torre et al., 2010]. We compared DCD and LI against Duet on the benchmarks provided by the authors of these tools. The programs were generated by SatAbs from a set of Linux device drivers. The input formats of the tools implementing DCD and LI are incompatible, so we report the results separately. The `ib700wdt` and `mixcomwd` benchmarks were generated from the same device drivers, but refer to a different set of Boolean programs in Tables 3.2 and 3.3. All benchmarks were run with a timeout of 5 minutes.

Linear interfaces Table 3.2 presents the results of comparison with Linear Interfaces (LI) on the set of Boolean programs reported in [La Torre et al., 2010]. Each LI benchmark consists of a server and a client thread template, where the client template is replicated arbitrarily many times. The client thread template is the device driver code, and the server thread template simulates the OS interacting with the drivers.

LI is an under approximation method based on limiting the number of scheduler rounds (as opposed to context-switches) and the use of *linear interfaces* to summarize interference in a round. In the LI implementation, the system is tested under 4 rounds of scheduling to look for a counter example, and if one is not found then an adequacy checker is executed that *may* succeed in proving the program safe for arbitrarily many threads and rounds of scheduling. The *safe* columns refer to the number of instances that were proved safe (for each analysis). The *unsafe* column for LI refers to the instances for which LI found a counterexample (a confirmed bug), while in Duet, it refers to the instances where assertions could not be proved safe. Note that since our approach is not complete, failure to prove an assertion does not imply that the assertion is necessarily false. The *timeout* column for LI refers to instances where LI

cannot finish checking the program under 4 rounds, or cannot find a counterexample under 4 rounds and the adequacy checker times out while trying to prove the program safe. The *timeout* column for Duet refers to all the instances that Duet cannot prove safe within the timeout limit. The *unknown* column for LI refers to the instances that no counterexample is found, and the adequacy checker finishes but fails to prove the program safe for arbitrary number of threads. On the LI set of benchmarks, Duet's octagon analysis configuration outperforms interval analysis. Both of Duet's configurations are able to prove more instances safe than LI.

As with any static analysis tool, there are programs for which Duet reports an error where there is none (a *false alarm*). The table below gives us insight into how often this occurs. The table presents a break-down of the outcomes of LI and Duet's Octagon Analysis configuration across all benchmarks. The rows of the table correspond to the possible outcomes of Duet, and the columns to the possible outcomes of LI. The number in cell (i, j) in the table indicates the number of programs such that the outcome of Duet is i and the outcome of LI is j. There are 60 instances for which Duet reports a *confirmed false alarm* (a program that LI proves to be safe, but Duet fails to prove safe) and 247 instances for which Duet reports a bug that is confirmed by LI. This allows us to estimate the false alarm rate at 20% ($= 60/(247 + 60)$). Another interesting fact that we can observe from this table is that there there are few programs that LI can prove safe that Duet cannot (60+2), and many programs that Duet can prove safe that LI cannot (267+916).

		LI			
		safe	unsafe	timeout	unknown
Duet	safe	1320	0	267	916
	unsafe	60	247	25	538
	timeout	2	1	0	40

Dynamic cutoff detection Table 3.3 presents the results of comparison with the DCD algorithm on the set of Boolean programs used in [Kaiser et al., 2010]. In the DCD benchmarks, there is a single thread template that is replicated infinitely many times.

Dynamic cutoff detection [Kaiser et al., 2010] is a decision procedure for thread-state reachability in Boolean programs with infinitely many threads. The DCD algorithm explores finite-thread instantiations of the Boolean program with and iteratively increases the number of threads until a counterexample or a cutoff is found (a cutoff is a number of threads n such every thread state that is reachable with $m \geq n$ threads is also reachable with n threads).

For the subset of these benchmarks where DCD does not time out, the cutoff is at most 3 threads. Duet's interval and octagon analysis substantially outperforms DCD in proving programs correct. As with the integer experiments, interval analysis outperforms octagon analysis, but there are 3 instances where octagon analysis succeeds and interval analysis fails. With the combination of both configurations, Duet can prove a total of 58 programs correct (in contrast to 19 for DCD) and there are no programs that DCD proves safe and that Duet cannot.

3.8 Related Work

Concurrent program analysis Every program analysis relies on an *induction principle* to reason about unbounded computations. Program analyses for concurrent programs typically employ induction

principles derived from logics for concurrent programs. A general recipe for defining a program analysis is to take a program logic (which defines a system of verification conditions for *checking* a proof), interpret the verification conditions as a system of constraints on a hypothetical annotation, and to develop an algorithm to find a solution to those constraints [Cousot and Cousot, 1984]. In this sense, most concurrent program analyses can be understood as descendants of one of three program logics: [Ashcroft and Manna, 1970], [Owicki and Gries, 1976], and [Jones, 1981].

Ashcroft and Manna developed the first program logic for concurrent programs [Ashcroft and Manna, 1970]. The logic adapts Floyd's logic [Floyd, 1967] to concurrent programs (with finitely many threads) by "compiling" all the threads into a sequential program which represents the interleaved behaviour of all threads. From the program analysis perspective, this strategy yields very accurate program analyses, but comes at a high computational cost: the size of the interleaved sequential program is exponential in the number of threads.

The program analysis presented in [Kahlon et al., 2009] is conceptually rooted in Ashcroft and Manna's logic. [Kahlon et al., 2009] improves upon the basic idea by representing the interleaved sequential behaviour of all threads by a *transaction graph*. The transaction graph is a more succinct representation of the interleaved behaviour of threads which suppresses redundant interleavings, in the style of partial order reduction [Valmari, 1991, Peled, 1993, Godefroid, 1994]. The constraints induced by the transaction graph are solved using a series of increasingly precise abstract domains. At each step, the results of the analysis are used to remove unreachable sections of the transaction graph, so that the next step (which uses a more computationally expensive abstract domain) operates on a smaller graph. Transaction graphs are a more precise representation of program behaviour than the data flow graphs presented in this chapter. However, transaction graphs for programs with infinitely many threads are infinite (and so do not yield an effective analysis), and are worst-case exponential in the number of threads. Data flow graphs are polynomial in the number of threads in the program.

Thread-modular program logics avoid the exponential explosion by reasoning about the sequential behaviour of each thread independently instead of their interleaved behaviour [Owicki and Gries, 1976, Jones, 1981]. To make this reasoning sound, proofs are required to satisfy a set of *non-interference* conditions which ensure that the sequential reasoning is robust under the action of other threads. From the perspective of program analysis, thread-modular logics gain scalability by abstracting away relationships between threads.

Owicki and Gries [Owicki and Gries, 1976] pioneered thread-modular program logics and introduced a *state-based* non-interference condition. The condition is that the annotation at each location in a thread is invariant under the action of every instruction of every *other* thread.

[Berdine et al., 2008] presents a technique for generating invariants of the shape "for all threads t, $\varphi(t)$ holds."[1] The technique can be thought of as a variation of the non-interference rule of Owicki-Gries, where every thread must additionally not interfere with itself. This makes the induction principle sound no matter how many threads are executing in parallel. Berdine et al. apply their construction to a shape domain and successfully verify linearizability for a number of concurrent data structures.

Like DFGs, [Berdine et al., 2008] generates thread-state invariants, which refer only to the local variables of one thread. One strategy for generating invariants which correlate local variables of different threads is *Thread correlation analysis* [Segalov et al., 2009]. Thread correlation analysis extends [Berdine et al., 2008] by deriving a *correlation invariant*, a formula over the globals and the local variables

[1] A similar class of invariants was investigated by Namjoshi in the context of finite-state processes [Namjoshi, 2007].

of two threads, which holds for every pair of threads. Reflective abstraction is another technique for generating invariants that are indexed over the local variables of more than one thread [Sanchez et al., 2012]. The idea is a hybrid of Ashcroft and Manna's proof system and the thread quantification approach. Given some number $k \in \mathbb{N}$, reflective abstraction builds a constraint system that is composed of k *materialized* threads, as well as one *mirror* thread representing the environment. The mirror thread serves a similar role to the self-interference condition in [Berdine et al., 2008].

From the stand-point of automated reasoning the inference check of [Owicki and Gries, 1976] is computationally expensive since it induces a $O(|\mathsf{Loc}| \times |\Sigma|)$ constraints to ensure the condition that the annotation at every location is invariant under every command. [Jones, 1981] avoids this expense by using *environment abstractions*. Rather than check interference-freedom with every command, we may perform the interference-freedom check with respect to the environment abstraction which approximates the effect of transitions on the global state. From the program analysis perspective, this loses precision (since the interference-freedom conditions are approximated by "abstract" interference-freedom conditions), but gains scalability.

[Miné, 2011] employs an environment abstraction which represents the collection of values that might be written to each variable during a thread's execution (e.g., the environment abstraction might represent "`x` might be assigned any value between 0 and 100"). This results in a highly scalable analysis, but does not allow for synchronization to be treated as precisely as in data flow graphs. Later work uses *relational* environment abstractions, which approximate the transition formula of a set of commands, rather than collecting the values the command may write (e.g., the environment abstraction can represent a relationship such as "the environment may increase the value of `x`") [Miné, 2014].

With the exception of [Kahlon et al., 2009], the above techniques employ a fixed induction principle which is determined by the syntax of the program. The method presented in this chapter and the one of [Kahlon et al., 2009] are the only ones we are aware of that *compute* an induction principle. Our method starts with an *unsound* induction principle (the sequential data flow graph) and gradually *coarsens* it until the induction principle is sound. In contrast, [Kahlon et al., 2009] starts with a *sound* induction principle and gradually *refines* it.

Data flow graphs Variations of data flow graphs have a long history within the compilers community, both as a means for exposing parallelism in sequential code for parallelizing compilers [Kuck et al., 1981, Ferrante et al., 1987], and as a data structure for use in sparse dataflow analysis [Weise et al., 1994, Johnson and Pingali, 1993].

Recently, DFGs have been used for generating invariants for sequential programs [Oh et al., 2012]. The attraction of data flow graphs in this setting is sparsity: using a data flow graph, it is possible to infer invariants over the "relevant" variables at every program location, rather than all program variables. This may have substantial advantages in terms of the time and memory consumption required for an analysis. [Oh et al., 2012] shows that for sequential data flow graphs (constructed using a reaching definitions analysis), running a particular type of independent attribute analysis yields the same results as on the control flow graph. [Oh et al., 2012] uses a variable packing technique similar to the one reported in Section 3.6.1 to support relational analyses.

Device Drivers	#programs	LI				Duet: Octagon Analysis			Duet: Interval Analysis		
		safe	unsafe	unknown	timeout	safe	unsafe	timeout	safe	unsafe	timeout
i8xx_tco	338	214	14	0	110	259	79	0	198	140	0
ib700wdt	181	109	13	0	59	124	56	1	91	90	0
machzwd	255	56	24	94	81	182	70	3	148	105	2
mixcomwd	178	103	24	0	51	117	59	2	81	95	2
pcwd	100	81	1	0	18	74	16	10	44	50	6
sbc60xxwdt	174	92	23	0	59	113	60	1	79	94	1
sc1200wdt	247	138	13	0	96	178	67	2	138	107	2
sc520_wdt	186	15	23	97	51	123	61	2	89	95	2
smsc37b787_wdt	340	154	13	0	173	272	65	3	151	187	2
w83877f_wdt	230	15	23	97	95	150	77	3	98	128	4
w83977f_wdt	389	147	13	0	229	322	65	2	144	243	2
wdt	230	109	17	0	104	161	59	10	108	114	8
wdt977	351	139	13	0	199	282	67	2	132	218	1
wdt_pci	217	10	34	4	169	146	69	2	146	69	2
total	3416	1382	248	292	1494	2503	870	43	1647	1735	34

Table 3.2: Comparison with linear interfaces [La Torre et al., 2010] for Boolean Programs. Average time per benchmark was 16.9s for LI and 3.4s for Duet. Benchmarks were run on an 3.16GHz Intel® Core 2™ machine with 4GB of RAM.

Device Drivers	#programs	DCD			Duet: Octagon Analysis			Duet: Interval Analysis		
		safe	unsafe	timeout	safe	unsafe	timeout	safe	unsafe	timeout
ib700wdt	132	10	102	20	16	113	3	28	101	3
mixcomwd	138	9	108	21	16	118	4	27	107	4

Table 3.3: Comparison with dynamic cutoff detection (DCD) [Kaiser et al., 2010] for Boolean programs. Average time per benchmark was 24.9s for DCD and 8.2s for Duet. Benchmarks were run on an 800MHz AMD® Opteron™ machine with 32 GB of RAM.

Part III

Software Model Checking

Introduction

Part II dealt with the problem of invariant generation: how to synthesize a property (ideally, the strongest possible) that is satisfied by all reachable states. Once a program invariant has been generated, it may be used to verify program properties by checking that the invariant implies the property of interest. However, if the invariant does *not* imply the property of interest, no conclusion may be drawn. Invariant generation can be used to *verify* properties, but not *refute them*.

Part III describes techniques for verifying and refuting properties of concurrent programs. The properties of interest are *non-reachability* properties: we wish to prove that the program never enters some designated error state. Many properties of interest (e.g., assertion checking or mutual exclusion) can be formulated as non-reachability properties. The techniques in this part are *complete for refutation* in the sense that if the error is reachable, they will produce a certificate that shows how the error can be reached. Since reachability is generally undecidable, the necessary trade-off for refutation completeness is that the techniques are potentially non-terminating. We use the term *software model checkers* to refer to the class of techniques that sacrifice termination for refutation-completeness.

Chapter 4 presents a software model checking technique for concurrent programs in which there are finitely many threads. Chapter 5 presents a generalization that also applies to programs with infinitely many threads. Both chapters are based on the *trace abstraction* paradigm for software model checking, introduced in [Heizmann et al., 2009]. In this paradigm, a program is associated with a language of error traces, namely all those traces that reach some designated error location, and the verification problem is to prove that every error trace is infeasible (does not correspond to an execution). In the remainder of this introduction, we will describe some common background for both chapters.

Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ be a program, and let ℓ_{err} be a control location. Define the set of *error traces* for a program P and error location ℓ_{err} as $\mathcal{L}(P, \ell_{\mathsf{err}}) \triangleq \{\tau \in \mathcal{L}(P) : \exists \mathtt{i} \in N.\mathsf{tgt}(\tau, \mathtt{i}) = \ell_{\mathsf{err}}\}$. Clearly, ℓ_{err} is reachable (in the sense that there is a reachable state $\langle \rho, loc \rangle$ and some thread $\mathtt{i} \in N$ such that $loc(\mathtt{i}) = \ell_{\mathsf{err}}$) if and only if some error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$ is feasible. Trace abstraction formulates the non-reachability verification problem as: to prove that ℓ_{err} is unreachable, synthesize a language **infeasible** traces that contains $\mathcal{L}(P, \ell_{\mathsf{err}})$.

The problem of constructing a language of infeasible traces that contains all error traces can be approached using a language learning strategy, depicted in Figure 3.6. The idea is to learn a proof from examples, by sampling error traces, proving them to be infeasible, and then assembling these proofs into an argument that every error trace is infeasible. The procedure iteratively constructs a proof object IP (IP is an *inductive data flow graph* in Chapter 4 and a *proof space* in Chapter 5) that represents a language of infeasible traces denoted $\mathcal{L}(IP)$, which is initially empty ($\mathcal{L}(IP) = \emptyset$). The procedure begins by sampling an error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$. If τ is feasible, then τ serves as a certificate that ℓ_{err} is reachable. If τ is *in*feasible, we construct an infeasibility proof IP_τ for τ (i.e., so that $\tau \in \mathcal{L}(IP_\tau)$).

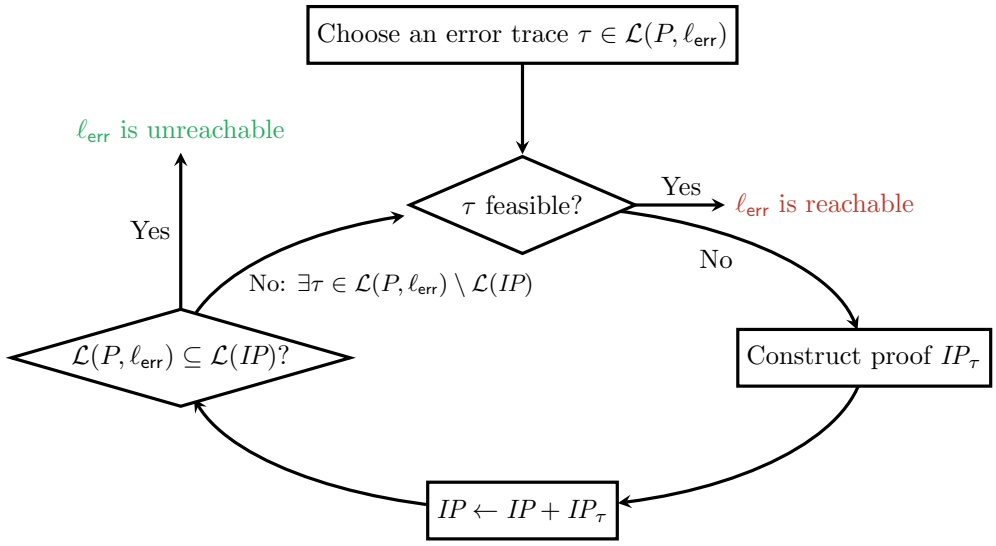


Figure 3.6: Verification via language learning

IP_τ is added to the accumulating infeasibility proof object IP_τ, resulting in a proof object that proves infeasibility of *at least* all traces in IP and IP_τ. Then we check whether every error trace is proved infeasible by IP ($\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(IP)$). If so, then IP is a certificate that ℓ_{err} is *unreachable* and the procedure terminates. If not, we may sample an *unproven* error trace in the difference $\mathcal{L}(P, \ell_{\mathsf{err}}) \setminus \mathcal{L}(IP)$, and begin the loop again. If this procedure terminates, it terminates in a certificate that either verifies or refutes the reachability of ℓ_{err}. The loop may repeat forever, failing to learn an infeasibility argument that generalizes to every error trace.

There are two interesting features of trace abstraction in the setting of concurrent programs. First, the proof IP is constructed by assembling proofs IP_τ that are obtained from traces. Although it may be challenging to formalize a correctness argument for a multi-threaded program, it is relatively easy to prove correctness of a trace. In a trace, all scheduling non-determinism is resolved making it effectively a simple straight-line sequential program. In this sense, trace abstraction is able to leverage the wealth of techniques available for synthesizing correctness proofs for *sequential* programs to reason about *concurrent* ones.

The second interesting feature of trace abstraction for concurrent programs is that we may design proof systems so that proof checking (the problem of testing $\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(IP)$) is *combinatorial*, in the sense that does not involve logical reasoning (i.e., reasoning about the values of program variables). Finding a trace in $\mathcal{L}(P, \ell_{\mathsf{err}}) \setminus \mathcal{L}(IP)$ (or conversely, verifying $\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(IP)$) is a problem that can be solved using exhaustive search. This kind of exhaustive search is the traditional strength of automata and (finite-state) model checking.

Learning proofs from examples relies on the success of proof generalization. That is, we need a plausible answer to the question *how do we take a single infeasible trace and synthesize a proof that applies to many traces?* We turn to our central thesis of *parallel proof for parallel programs*: the proof objects presented in the next two chapters aim to *parallelize* sequential proofs of traces so that they apply to many traces that are infeasible "for the same essential reason." By exposing the parallel structure of proofs, we obtain proof objects that are both *succinct* and *general*.

Chapter 4

Inductive Data Flow Graphs

This chapter introduces *inductive data flow graphs* (iDFGs), a proof system for reasoning about concurrent programs with a finite number of threads. Like data flow graphs, inductive data flow graphs explicitly represent the parallelism in a program, allowing for succinct proofs. Inductive data flow graphs can be used as a basis for software model checking, following the trace abstraction paradigm [Heizmann et al., 2009].

4.1 Overview

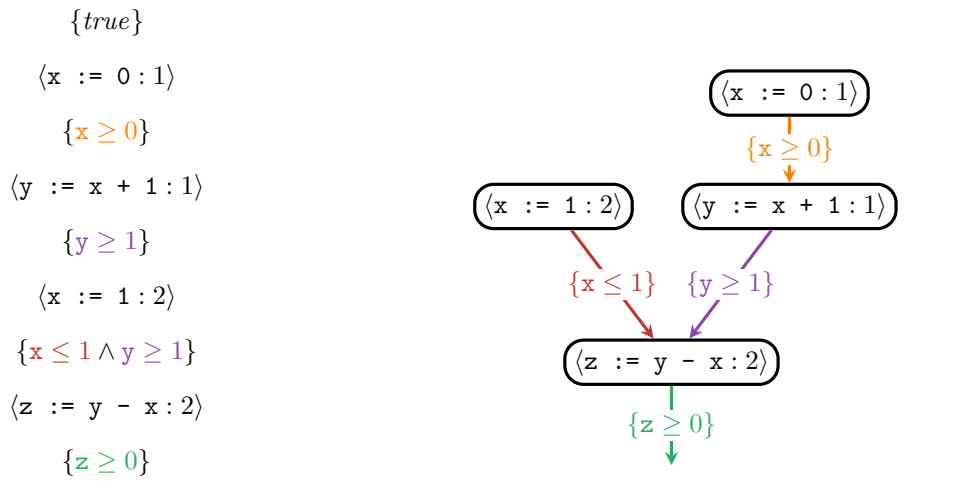
Just as we contrasted control flow graphs with data flow graphs in Section 3.1, we may contrast Floyd/Hoare proofs with inductive data flow graphs. Figure 4.1 depicts two proofs of the validity of the Hoare triple

$$\{\mathit{true}\}\ \langle \texttt{x := 0} : 1\rangle\langle \texttt{y := x + 1} : 1\rangle\langle \texttt{x := 1} : 2\rangle\langle \texttt{z := y - x} : 2\rangle\ \{z \geq 0\}\ .$$

One is a Floyd/Hoare proof and the other is an iDFG. The Floyd/Hoare proof is a sequence of intermediate assertions that show how the state of the program evolves over the trace. Events in the trace are linearly ordered in time and this linear order is reflected in the proof: first $\langle \texttt{x := 0} : 1\rangle$ is executed, after which $\texttt{x} \geq 0$ holds; then $\langle \texttt{y := x + 1} : 1\rangle$ is executed, after which $\texttt{y} \geq 1$ holds; then $\langle \texttt{x := 1} : 2\rangle$ is executed; after which $\texttt{x} \leq 1 \wedge \texttt{y} \geq 1$ holds; then $\langle \texttt{z := y - x} : 2\rangle$ is executed, after which $\texttt{z} \geq 0$ holds.

Much like a data flow graph presents a parallel view of a program, an iDFG presents a parallel view of a *proof*. Rather than edges representing matching read/write relationships as they do in a data flow graph, edges in an iDFG represent matching satisfaction and use of some proof obligation. Events are arranged in a partial order: the proof guarantees the post-condition $\texttt{z} \geq 0$ holds *after* executing $\langle \texttt{z := y - x} : 2\rangle$ on the condition that both $\texttt{x} \leq 1$ and $\texttt{y} \geq 1$ hold *before* executing $\langle \texttt{z := y - x} : 2\rangle$, but the order in which these two pre-conditions are achieved is irrelevant. The precondition $\texttt{x} \leq 1$ can be achieved by executing $\langle \texttt{x := 1} : 2\rangle$, and the precondition $\texttt{y} \geq 1$ can be achieved by executing $\langle \texttt{y := x + 1} : 1\rangle$ in a state where $\texttt{x} \geq 0$ holds (which, in turn, can be achieved by executing $\langle \texttt{x := 0} : 1\rangle$).

Since iDFGs suppress irrelevant ordering constraints, the correspondence between Floyd/Hoare proofs and iDFGs is many-to-one. Phrased differently, an iDFG can be seen as a representation of a *set* of traces (all of which have the same iDFG proof). The importance of this is that inductive data flow graphs serve as a mechanism for *proof generalization*: given a proof of a single trace τ, by "*parallelizing*"



(a) Floyd/Hoare proof of a trace

(b) Inductive Data Flow Graph for the same trace

Figure 4.1: A Floyd/Hoare proof and iDFG for the same Hoare triple.

the proof we construct an iDFG that proves a property of interest about τ and also many other traces. This is important for two reasons: first, because it yields *succinct* proof objects (there is a formal sense in which iDFG proof are small), and second because it makes the trace abstraction algorithm pictured in Figure 3.6 converge more quickly.

A crucial question is *when is an iDFG a proof of some non-reachability property?* In view of the trace abstraction paradigm, *how can we mechanically check that all error traces "belong" to an iDFG proof*? First, we must clarify what it means for a trace to belong to an iDFG. Each edge in the iDFG in Figure 4.1b can be seen as an ordering constraint. For example, the edge

$$\langle \texttt{x := 1 : 2} \rangle \xrightarrow{\{x \leq 1\}} \langle \texttt{z := y - x : 2} \rangle$$

represents the constraint that $\langle \texttt{x := 1 : 2} \rangle$ happens *before* $\langle \texttt{z := y - x : 2} \rangle$ in the trace, and every command in between leaves the assertion $\texttt{x} \leq 1$ invariant. A trace that satisfies all the ordering constraints of an iDFG is said to be *recognized* by it. Figure 4.2 pictures three traces recognized by this iDFG, overlaid with the ordering constraints. Notice that the iDFG recognizes different interleavings of the same trace (the second and third commands are swapped in the middle trace), and also recognizes traces that contain commands that are irrelevant to the proof ($\langle \texttt{z := 0 : 3} \rangle$ in the right-most trace).

The characteristic that distinguishes formal program logics from ad-hoc reasoning is that a proof in a program logic can be verified mechanically. Verification of iDFG "proofs" must be mechanical to justify calling them as such. In particular, we must be able to verify that all traces of a program that violate some non-reachability property are recognized by an iDFG. We show that there is a natural way to encode the proof checking problem as an inclusion problem between alternating finite automata, which can be solved in polynomial space.

The organization of this chapter is as follows:

- Section 4.2 defines inductive data flow graphs and formulates an iDFG-based proof rule for verifying non-reachability properties.
- Section 4.3 shows how the premise of the proof rule can be checked mechanically (in polynomial

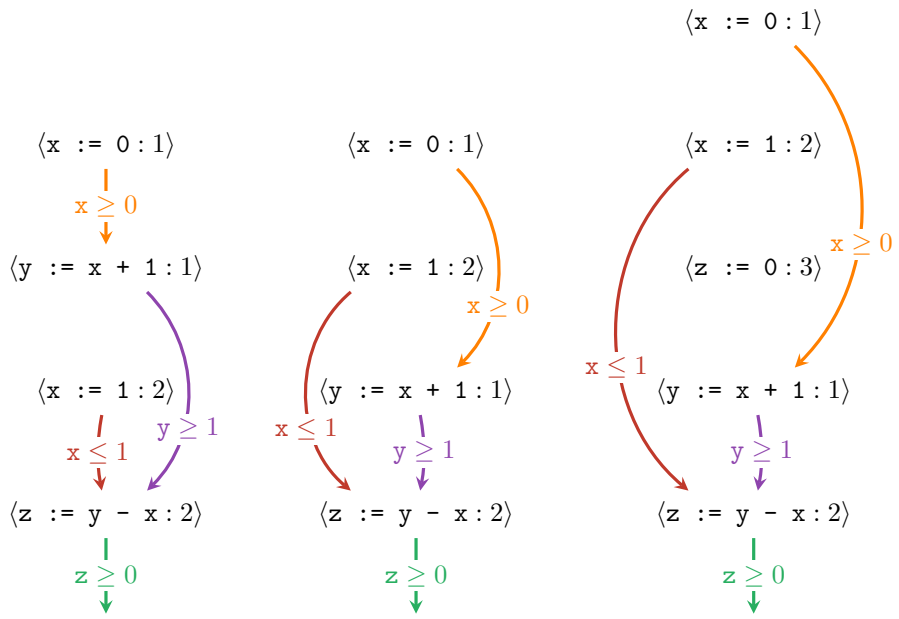


Figure 4.2: Traces recognized by the iDFG in Figure 4.1b

space) by reducing it to an inclusion problem for alternating finite automata.

- Section 4.4 justifies the claim that iDFGs are succinct proof objects. We show that if there exists a small proof of an non-reachability property, then there exists a small iDFG proof.
- Section 4.5 shows how to automate the proof rule of Section 4.2, following the trace abstraction paradigm.
- Section 4.6 compares inductive data flow graphs with related work.

4.2 Inductive Data Flow Graphs (iDFGs)

This section defines inductive data flow graphs and their associated proof rule.

Definition 4.2.1 (Inductive Data Flow Graph, iDFG)**.** *Let $P = \langle \mathit{Loc}, \Sigma, \ell_{\mathit{init}}, N \rangle$ be a program. An* inductive data flow graph *(iDFG) for P is a tuple $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathit{post}} \rangle$ where*

- *V is a set of vertices.*
- *$E \subseteq V \times \mathit{Formula}(\mathit{GV} \cup \mathit{LV}(N)) \times V$ is a set of edges labelled by program assertions. We use the notation $u \xrightarrow{\varphi} v$ to denote an edge from u to v labelled by the assertion φ. For a vertex $v \in V$, we use*

$$\mathit{in}_G(v) \triangleq \{\varphi : \exists u \in V. u \xrightarrow{\varphi} v \in E\}$$
$$\mathit{out}_G(v) \triangleq \{\psi : \exists w \in V. v \xrightarrow{\psi} w \in E\}$$

to denote its sets of incoming and outgoing edge labels, respectively. We omit the G subscript if it can be inferred from the context.

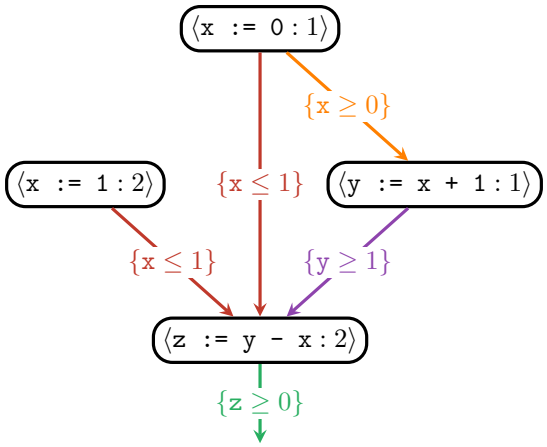


Figure 4.3: A simple iDFG

- $\mathit{cmd} : V \to \Sigma(N)$ *is a function that labels every vertex with an indexed command of* P.
- $\varphi_{\mathit{post}} \in \mathit{Formula}(\mathit{GV} \cup \mathit{LV}(N))$ *is an assertion called the* post-condition *of* G.

Furthermore, the labelling of an iDFG must be inductive *in the sense that:*

1. Consecution: *For every* $v \in V$*, and every* $\psi \in \mathsf{out}(v)$*, the Hoare triple* $\{\bigwedge \mathsf{in}(v)\}\ \mathit{cmd}(v)\ \{\psi\}$ *is valid, and*
2. Safety: *For every* $v \in V_{\mathsf{final}}$*, the Hoare triple* $\{\bigwedge \mathsf{in}(v)\}\ \mathit{cmd}(v)\ \{\varphi_{\mathit{post}}\}$ *is valid.*

Let G be an iDFG with post-condition φ_{post}. G can be associated with a set of traces τ for which G serves as a proof that $\{\mathit{true}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid ("φ_{post} is a post-condition of τ"). Since our aim is to prove *infeasibility* of traces, iDFGs with post-condition *false* are of particular interest; however, allowing an arbitrary post-condition in the general definition will be useful for induction arguments on iDFGs.

Consider the iDFG pictured in Figure 4.3. The neighbourhood of the vertex $\langle \texttt{z := y - x} : 2\rangle$ can be interpreted as a statement that if both $\texttt{x} \leq 1$ and $\texttt{y} \geq 1$ hold *before* executing $\langle \texttt{z := y - x} : 2\rangle$, then $\texttt{z} \geq 0$ holds *after*. There are two ways of achieving the pre-condition $\texttt{x} \leq 1$ (corresponding to the two incoming edges both labelled $\texttt{x} \leq 1$) and one way of achieving the pre-condition $\texttt{y} \geq 1$. This statement can be interpreted as a constraint on traces recognized by the iDFG. Multiple *inputs* to a vertex (the set $\mathsf{in}(v)$) are interpreted conjunctively (all incoming pre-conditions may hold), while multiple incoming edges with the same label (the set $\{u : u \xrightarrow{\varphi} v \in E\}$) are interpreted disjunctively (at least one path to achieving the pre-condition φ must be realized). The formal definition of the set of traces recognized by an iDFG follows this intuition.

Definition 4.2.2. *Let* $G = \langle V, E, \mathit{cmd}, V_{\mathsf{final}}, \varphi_{\mathit{post}}\rangle$ *be an iDFG for a program* $P = \langle \mathit{Loc}, \Sigma, \ell_{\mathit{init}}, N\rangle$*. Define a function* $\mathcal{L}_G : V \times \mathit{Formula}(\mathit{GV} \cup \mathit{LV}(N)) \to 2^{\Sigma(N)^*}$ *mapping iDFG vertices and formulas to trace languages to be the least function (in point-wise subset inclusion ordering) such that* $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}_G(v, \varphi)$ *if and only if one of the following hold:*

1. $\{\varphi\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi\}$ *is valid and* τ *belongs to* $\mathcal{L}_G(v, \varphi)$*, or*

2. $\mathit{cmd}(v) = \langle \sigma : \mathtt{i} \rangle$ *and for all* $\psi \in \mathit{in}(v)$ *there exists some edge* $u \xrightarrow{\psi} v \in E$ *such that* τ *belongs to* $\mathcal{L}_G(u, \psi)$.

Define $\mathcal{L}(G)$*, the set of traces recognized by* G*, to be* $\mathcal{L}(G) \triangleq \bigcup_{v \in V_{\mathsf{final}}} \mathcal{L}_G(v, \varphi_{\mathit{post}})$.

The foundational soundness theorem concerning iDFGs is that every trace recognized by an iDFG satisfies its post-condition:

Proposition 4.2.3. *Let* $G = \langle V, E, \mathit{cmd}, V_{\mathsf{final}}, \varphi_{\mathit{post}} \rangle$ *be an iDFG. For every trace* $\tau \in \mathcal{L}(G)$*, the Hoare triple* $\{\mathit{true}\}\ \tau\ \{\varphi_{\mathit{post}}\}$ *is valid.*

Proof. Let $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathsf{post}} \rangle$ be an iDFG. We prove that any vertex v, any assertion φ such that $\{\bigwedge \mathsf{in}(v)\}\ \mathsf{cmd}(v)\ \{\varphi\}$ is valid, and for any trace $\tau \in \mathcal{L}_G(v, \varphi)$, we have that $\{\mathit{true}\}\ \tau\ \{\varphi\}$ is valid. The main result follows, since $\tau \in \mathcal{L}(G)$ if and only if $\tau \in \mathcal{L}_G(v, \varphi_{\mathsf{post}})$ for some $v \in V_{\mathsf{final}}$, and $\{\bigwedge \mathsf{in}(v)\}\ \mathsf{cmd}(v)\ \{\varphi\}$ is valid by the safety condition for iDFGs.

We proceed by induction on τ.

- Base case $\tau = \epsilon$. This case is vacuous, because ϵ does not belong to $\mathcal{L}_G(v, \varphi)$ for any v or φ.
- Induction step $\tau = \tau' \langle \sigma : \mathtt{i} \rangle$. Following Definition 4.2.2, there are two cases to consider:
  1. $\{\varphi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid and $\tau' \in \mathcal{L}_G(v, \varphi)$. By the induction hypothesis, we have that $\{\mathit{true}\}\ \tau'\ \{\varphi\}$ is valid. Sequential composition with $\{\varphi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ yields the validity of $\{\mathit{true}\}\ \tau\ \{\varphi\}$.
  2. $\mathsf{cmd}(v) = \langle \sigma : \mathtt{i} \rangle$ and for all $\psi \in \mathsf{in}(v)$ there exists some u such that $u \xrightarrow{\psi} v \in E$ and $\tau' \in \mathcal{L}_G(u, \psi)$. Observe that for any $u \in V$ and assertion ψ such that $u \xrightarrow{\psi} v \in E$, we have $\{\bigwedge \mathsf{in}(u)\}\ \mathsf{cmd}(u)\ \{\psi\}$ by the consecution condition for iDFGs. Thus, for all $\psi \in \mathsf{in}(v)$, we have that $\{\mathit{true}\}\ \tau'\ \{\psi\}$ is valid by the induction hypothesis. By conjunction rule of Hoare logic, $\{\mathit{true}\}\ \tau'\ \{\bigwedge \mathsf{in}(v)\}$ is valid. Since $\{\bigwedge \mathsf{in}(v)\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid by assumption, we infer the validity of $\{\mathit{true}\}\ \tau\ \{\varphi\}$ by sequential composition. □

Finally, we conclude this section with a theorem that summarizes the proof rule associated with iDFGs: to prove that a program satisfies a non-reachability property, it is sufficient to exhibit an iDFG G with post-condition *false* such that every program trace violating the property is recognized by G.

Theorem 4.2.4. *Let* $P = \langle \mathit{Loc}, \Sigma, \ell_{\mathit{init}}, N \rangle$ *be a program and let* $\ell_{\mathit{err}} \in \mathit{Loc}$ *be a location. If there exists an iDFG* G *with post-condition false such that* $\mathcal{L}(P, \ell_{\mathit{err}}) \subseteq \mathcal{L}(G)$*, then* ℓ_{err} *is unreachable.*

4.3 Checking iDFGs

This section investigates the question of how one may algorithmically check the premise of the proof rule suggested by Theorem 4.2.4. That is, given a program P, an error location ℓ_{err}, and an iDFG G, how can we mechanically verify the inclusion $\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(G)$? We reduce this check to an inclusion problem between two alternating finite automata, which can be solved in polynomial space.

Alternating finite automata (AFA) [Chandra et al., 1981, Brzozowski and Leiss, 1980] are a class of finite automata that exhibit both existential (angelic, disjunctive) and universal (demonic, conjunctive) choice. Upon reading a letter of its input alphabet, an alternating finite automaton transitions from a

state to a positive Boolean formula over the states of the automaton: if the formula is a conjunction, the automaton must accept along *every* path; if the formula is a disjunction, the automaton must accept along *some* path. We recall the formal definition of alternating finite automata below.

Definition 4.3.1 (Positive Boolean Formula)**.** *For any set X, we use $\mathcal{B}^+(X)$ to denote the set of positive Boolean formulas over X (that is, Boolean formulas built from true, false, and the members of X using the binary connectives $\wedge$ and $\vee$).*

Definition 4.3.2 (Alternating finite automaton)**.** *An* alternating finite automaton *(AFA) is a tuple $A = \langle \Gamma, Q, \delta, \varphi_{\mathit{init}}, F \rangle$ where Γ is a finite alphabet, Q is a finite set of states, $\delta : Q \times \Gamma \to \mathcal{B}^+(Q)$ is a transition function that assigns every state q and letter $a \in \Gamma$ a positive Boolean formula over Q, φ_{init} (the* initial formula*) is a positive Boolean formula over Q, and $F \subseteq Q$ is a set of accepting states.*

Definition 4.3.3 (AFA semantics)**.** *Let $A = \langle \Gamma, Q, \delta, \varphi_{\mathit{init}}, F \rangle$ be an alternating finite automaton. Define a function $\mathcal{L}_A : \mathcal{B}^+(Q) \to 2^{\Gamma^*}$ mapping positive Boolean formulas over Q to languages over Γ recursively as follows:*

$$
\begin{aligned}
\epsilon \in \mathcal{L}_A(q) &\iff q \in F \\
aw \in \mathcal{L}_A(q) &\iff w \in \mathcal{L}_A(\delta(q, a)) \\
w \in \mathcal{L}_A(\varphi \wedge \psi) &\iff w \in \mathcal{L}_A(\varphi) \text{ and } w \in \mathcal{L}_A(\psi) \\
w \in \mathcal{L}_A(\varphi \vee \psi) &\iff w \in \mathcal{L}_A(\varphi) \text{ or } w \in \mathcal{L}_A(\psi) \\
w \in \mathcal{L}_A(\mathit{true}) &\iff \mathit{true} \\
w \in \mathcal{L}_A(\mathit{false}) &\iff \mathit{false}
\end{aligned}
$$

Define $\mathcal{L}(A)$, the language recognized by A, to be $\mathcal{L}(A) \triangleq \mathcal{L}_A(\varphi_{\mathit{init}})$.

4.3.1 Recognizing iDFG languages

The capacity of alternating finite automata to make both existential and universal choices makes them a natural fit for capturing the language recognized by an iDFG. Consider again the iDFG in Figure 4.3. The iDFG guarantees the post-condition $\texttt{z} \geq 0$ after executing $\langle \texttt{z := y - x} : 2 \rangle$ under the condition that $\texttt{x} \leq 1$ *and* $\texttt{y} \geq 1$ hold before (universal choice). The pre-condition $\texttt{x} \leq 1$ can be achieved by executing either $\langle \texttt{x := 1} : 2 \rangle$ *or* $\langle \texttt{x := 0} : 1 \rangle$ previously (existential choice), and the pre-condition $\texttt{y} \geq 1$ must be achieved by executing $\langle \texttt{y := x + 1} : 1 \rangle$ previously (and $\langle \texttt{x := 0} : 1 \rangle$ before that).

Comparing the definitions of the language recognized by an alternating finite automaton (Definition 4.3.3) and an inductive data flow graph (Definition 4.2.2), we see that an inductive data flow graph reads traces *backwards*. That is, we treat the incoming edges to an iDFG vertex as constraints on what happened *previously* in an execution, rather than reading its outgoing edges as constraints on what will happen in the future. For this reason, it is natural to define an AFA that recognizes the *reversal* of the language recognized by an iDFG. We use τ^{rev} to denote the reversal of a trace τ and L^{rev} to denote the reversal of a language L. Since our ultimate goal is to verify that every error trace is recognized by an iDFG G, we may verify instead that every *reversed* error trace is recognized by the AFA that recognizes $\mathcal{L}(G)^{rev}$. As we will see later in the section, it is possible to efficiently construct an AFA that recognizes reversed error traces, so that the verification problem is reduced to AFA inclusion.

Let $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathsf{post}} \rangle$ be an iDFG. We may construct an alternating finite automaton $\mathcal{A}(G) = \langle \Sigma(N), Q, \delta, q_0, Q_{\mathsf{final}} \rangle$ that recognizes $\mathcal{L}(G)^{rev}$ as follows.

- $Q \triangleq \{(v, \psi) : v \in V \wedge \psi \in \mathsf{out}(v)\} \cup \{(v_{\mathsf{f}}, \varphi_{\mathsf{post}}) : v_{\mathsf{f}} \in V_{\mathsf{final}}\}$. We will construct $\mathcal{A}(G)$ so that a trace τ is accepted starting from the state (v, ψ) (i.e., $\tau \in \mathcal{L}_{\mathcal{A}(G)}(v, \psi)$) if and only if $\tau^{rev} \in \mathcal{L}_G(v, \psi)$.
- $\delta((v, \psi), \langle \sigma : \mathtt{i} \rangle) \triangleq \mathit{skip}((v, \psi), \langle \sigma : \mathtt{i} \rangle) \vee \mathit{step}((v, \psi), \langle \sigma : \mathtt{i} \rangle)$, where *skip* and *step* correspond directly to cases 1 and 2 in Definition 4.2.2:

$$\mathit{skip}((v, \psi), \langle \sigma : \mathtt{i} \rangle) \triangleq \begin{cases} (v, \psi) & \text{if } \{\psi\}\ a\ \{\psi\} \text{ is valid} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{step}((v, \psi), \langle \sigma : \mathtt{i} \rangle) \triangleq \begin{cases} \bigwedge_{\varphi \in \mathsf{in}(v)} \bigvee_{u \xrightarrow{\varphi} v \in E} (u, \varphi) & \text{if } \mathsf{cmd}(v) = \langle \sigma : \mathtt{i} \rangle \\ \mathit{false} & \text{otherwise} \end{cases}$$

- $q_0 \triangleq \bigvee_{v_{\mathsf{f}} \in V_{\mathsf{final}}} (v_{\mathsf{f}}, \varphi_{\mathsf{post}})$ (initially, the automaton is in least one of the states corresponding to a final vertex of G).
- $Q_{\mathsf{final}} \triangleq \emptyset$ (there are no accepting states – the automaton accepts after transitioning to *true*).

This construction yields the following proposition.

Proposition 4.3.4. *Let $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathsf{post}} \rangle$ be an inductive data flow graph. There exists an alternating finite automaton $\mathcal{A}(G)$ that recognizes the* reversal *of the language of traces recognized by G, and which has at most $|E| + |V_{\mathsf{final}}|$ states.*

4.3.2 Recognizing error traces

The capability of alternating finite automata to make universal choices enables them to efficiently recognize the interleaved traces of a concurrent program. Intuitively, we may represent a control state of a concurrent program by a conjunction of states of an alternating finite automaton, with one conjunct for each thread.

Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ be a program (with N finite) and let ℓ_{err} be an error location. We construct an alternating finite automaton $\mathcal{A}(P, \ell_{\mathsf{err}}) = \langle Q, \Sigma(N), \delta, q_0, Q_{\mathsf{final}} \rangle$ that recognizes the reversal of the language of traces of P to ℓ_{err} as follows.

- $Q \triangleq \{\ell^{\mathtt{i}} : \ell \in \mathsf{Loc}, \mathtt{i} \in N\} \cup \{I\}$. We construct $\mathcal{A}(P, \ell_{\mathsf{err}})$ so that:
  - a trace τ is accepted starting from the state $\ell^{\mathtt{i}}$ (i.e., $\tau \in \mathcal{L}_{\mathcal{A}(P, \ell_{\mathsf{err}})}(\ell^{\mathtt{i}})$) iff $\tau|_{\mathtt{i}}$ corresponds to a reversed control flow path of P with $\mathsf{tgt}(\tau|_{\mathtt{i}}^{rev}, \mathtt{i}) = \ell$, where $\tau|_{\mathtt{i}}$ is the trace obtained from τ by deleting every command *not* executed by $\mathtt{i}$.
  - a trace τ is accepted starting from I (which we call the *initializer* state) iff τ^{rev} is an interleaved control flow path of P.
- The transition function δ follows the *reversed* commands:

$$\delta(\ell^{\mathtt{i}}, \langle \sigma : \mathtt{j} \rangle) \triangleq \begin{cases} \ell^{\mathtt{i}} & \text{if } \mathtt{i} \neq \mathtt{j} & \textit{(ignore commands of other threads)} \\ \mathsf{src}(\sigma)^{\mathtt{i}} & \text{if } \mathtt{i} = \mathtt{j} \wedge \mathsf{tgt}(\sigma) = \ell & \textit{(follow the edge } \sigma \textit{ backwards)} \\ \mathit{false} & \text{if } \mathtt{i} = \mathtt{j} \wedge \mathsf{tgt}(\sigma) \neq \ell & \textit{(reject if } \mathtt{i} \textit{ is not at } \mathsf{tgt}(\sigma)\textit{)} \end{cases}$$

and for the designated *initializer* I, we have

$$\delta(I, \langle \sigma : \mathtt{i} \rangle) \triangleq I \wedge \mathsf{src}(\sigma)^{\mathtt{i}}$$

(note that the automaton never leaves the state I once it enters)

- $q_0 \triangleq I \wedge \bigvee_{\mathtt{i} \in N} \ell_{\mathsf{err}}^{\mathtt{i}}$ (initially, the automaton is in state I *and* at least one thread is at the error state).
- $Q_{\mathsf{final}} \triangleq \{\ell_{\mathsf{init}}^{\mathtt{i}} : \mathtt{i} \in N\} \cup \{I\}$ (the automaton accepts when every thread is in the initial location).

This construction yields the following proposition.

Proposition 4.3.5. *Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N \rangle$ be a program with a finite number of threads, and let $\ell_{err} \in \mathsf{Loc}$ be a control location. There exists an alternating finite automaton $\mathcal{A}(P, \ell_{err})$ that recognizes the* reversal *of the language of error traces $\mathcal{L}(P, \ell_{err})$, and that has at most $|\mathsf{Loc}| + 1$ states.*

4.3.3 Mechanical verification of iDFG proofs

Finally, we conclude with the theorem that the problem of checking whether every error trace of a program is recognized by an iDFG can be decided in polynomial space.

Theorem 4.3.6. *Let P be a program, ℓ_{err} be an error location, and G be an iDFG. The problem of checking whether all error trace of P are recognized by G ($\mathcal{L}(P, \ell_{err}) \subseteq \mathcal{L}(G)$) is decidable in polynomial space.*

Proof. Let P be a program, ℓ_{err} be an error location, and G be an iDFG. The theorem follows directly from the AFA constructions

$$\begin{aligned} \mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(G) &\iff \mathcal{L}(P, \ell_{\mathsf{err}})^{rev} \subseteq \mathcal{L}(G)^{rev} \\ &\iff \mathcal{L}(\mathcal{A}(P, \ell_{\mathsf{err}})) \subseteq \mathcal{L}(\mathcal{A}(G)) \qquad \text{Proposition 4.3.5 \& Proposition 4.3.4} \end{aligned}$$

It is well known that the inclusion between two languages recognized by alternating finite automata can be decided in polynomial space. Since the constructions of both $\mathcal{A}(P, \ell_{\mathsf{err}})$ and $\mathcal{A}(G)$ can be performed in polynomial space, the theorem follows. □

4.4 Succinctness of iDFGs

This section justifies the claim that iDFGs are succinct proof objects. The result that we are after is that *if there is a small proof that a program satisfies some non-reachability property, then there is a small iDFG that proves all error traces are infeasible.* Making this statement precise requires formalizing a proof system in which to interpret the premise (what does it mean for there to be a small proof?). We introduce *localized proofs* to serve this purpose. The section culminates in a result that localized proofs can be translated into iDFGs with only a polynomial increase in size.

We begin by considering the question: *what does it mean for there to be a "small proof" of a given property?* Before we can tackle the question of what constitutes a *small* proof, we must define proofs. A first candidate definition is provided by Ashcroft and Manna's proof system [Ashcroft and Manna, 1970],

the first logic designed for reasoning about concurrent programs. This logic adapts Floyd's logic [Floyd, 1967] to the concurrent setting by "compiling" a concurrent program into a sequential one. This amounts to annotating every control state with an assertion subject to certain verification conditions. The definition of Ashcroft/Manna proofs (adapted to proving non-reachability properties of PLIP programs) is recalled below.

Definition 4.4.1 (Ashcroft/Manna proof, [Ashcroft and Manna, 1970])**.** *An* Ashcroft/Manna *proof for a program* $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N \rangle$ *and error location* ℓ_{err} *is a function* $\Phi^{AM} : \mathsf{ControlState} \to \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}(N))$ *satisfying the following three properties:*

- Initiation: $\Phi^{AM}(loc_{init}) = true$
- Consecution: *for any* $\sigma \in \Sigma$ *and any thread* $\mathtt{i} \in N$*, for any* $loc \in \mathsf{ControlState}$ *such that* $loc(\mathtt{i}) = \mathsf{src}(\sigma)$*, the following Hoare triple is valid:*

$$\{\Phi^{AM}(loc)\}\ \langle \sigma : \mathtt{i} \rangle\ \{\Phi^{AM}(loc[\mathtt{i} \leftarrow \mathsf{tgt}(\sigma)])\}\ .$$

- Safety: *for any* $loc \in \mathsf{ControlState}$ *such that* $loc(\mathtt{i}) = \ell_{err}$ *for some thread* $\mathtt{i} \in N$*, we have* $\Phi^{AM}(loc) = false$*.*

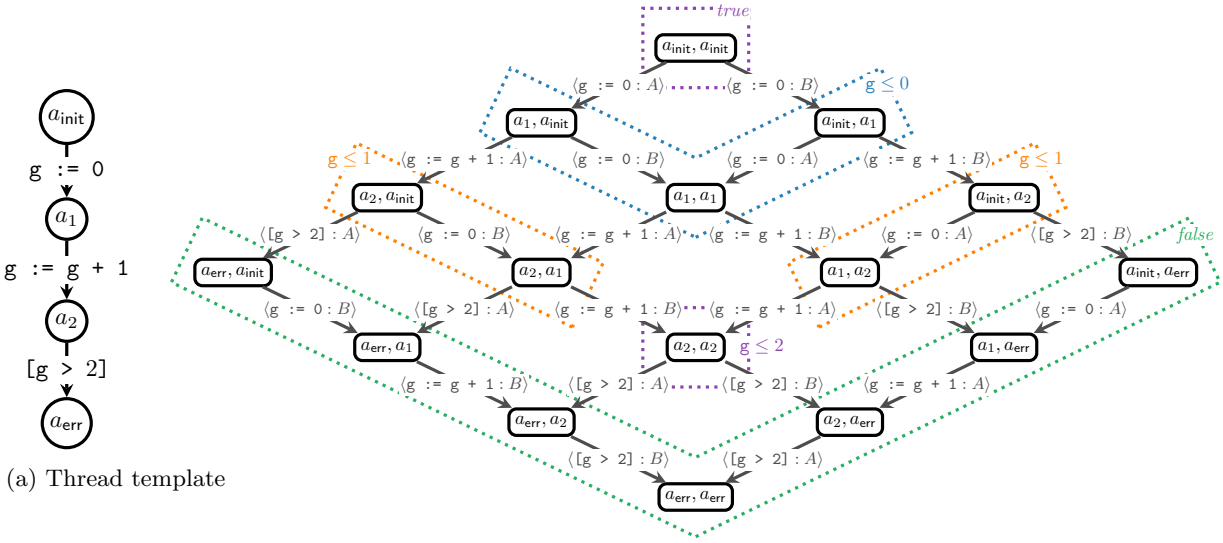
Example 4.4.2. *Figure 4.4 depicts a program that is executed by two threads, both of which set a global variable* g *to 0, increment* g*, and then go to an error location* a_{err} *if* g *is greater than two. A simple intuitive correctness argument is that the error* a_{err} *is unreachable because it is guarded by the command* `[g > 2]`*, but we always have* $\mathsf{g} \leq 2$ *because each thread contributes 1 to* g *and there are only 2 threads.*

A depiction of an Ashcroft/Manna proof appears in Figure 4.4. Notice that the proof is large, but also redundant. Every node is labelled with one of five assertions: $true$*,* $\mathsf{g} \leq 0$*,* $\mathsf{g} \leq 1$*,* $\mathsf{g} \leq 2$*, or* $false$*.*

Ashcroft and Manna's proof system is a natural choice for defining proofs, but a poor candidate for defining *small* ones. An Ashcroft/Manna annotation defines an assertion for each global control state in a program, of which there are only $|\mathsf{Loc}|^{|N|}$. However, Ashcroft/Manna proofs serve as a kind of *assembly language* for proofs, in the sense that proofs in other proof systems can be translated into Ashcroft/Manna proofs, typically at the cost of making the proof "larger" in some sense. This is the essential idea behind *localized proofs*, to be introduced later in this section: we re-use Ashcroft and Manna's proof system, but expose some additional structure so that translation into localized proofs does not blow up. Before we define localized proofs, we give *thread-modular proofs* an example of the kind of proof translation process we aim to capture.

Thread-modular proofs are motivated by the seminal work of Owicki and Gries [Owicki and Gries, 1976]. Rather than reason about the combined behaviour of all threads as a sequential program, thread-modular proofs reason about each thread individually. The soundness of thread-modular reasoning relies on an additional *interference-freedom* property that requires that proofs are robust under interference from other threads. We define thread-modular proofs as follows:

Definition 4.4.3 (Thread-modular proof)**.** *Let* $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N \rangle$ *be a program and let* $\ell_{err} \in \mathsf{Loc}$ *be an error location. A* thread-modular proof *of the non-reachability of* ℓ_{err} *is a family of maps* $\{\Phi^{TM}_{\mathtt{i}}\}_{\mathtt{i} \in N}$ *such that for each* $\mathtt{i} \in N$*,* $\Phi^{TM}_{\mathtt{i}} : \mathsf{Loc} \to \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}(N))$ *is a function mapping control locations of* P *to assertions over global variables and the local variables of all threads, and such that the following conditions are satisfied.*



(a) Thread template

(b) Ashcroft/Manna proof

Figure 4.4: An Ashcroft/Manna proof that a_{err} is unreachable in the program consisting of two threads that execute the thread template (a). The dotted boxes depict the mapping from control states to program assertions: each dotted box is attached to an assertion that is used to label every vertex appearing inside the box.

- Initiation: *for all* $\mathtt{i} \in N$, $\Phi^{TM}_{\mathtt{i}}(\ell_{init}) = \mathit{true}$
- Consecution: *for all* $\sigma \in \Sigma$ *and* $\mathtt{i} \in N$, *the Hoare triple* $\{\Phi^{TM}_{\mathtt{i}}(\mathsf{src}(\sigma))\}\ \langle \sigma : \mathtt{i} \rangle\ \{\Phi^{TM}_{\mathtt{i}}(\mathsf{tgt}(\sigma))\}$ *is valid*
- Interference-Freedom: *for all* $\sigma \in \Sigma$, *for all* $\mathtt{i} \neq \mathtt{j} \in N$, *for all* $\ell \in \mathsf{Loc}$, *the Hoare triple*

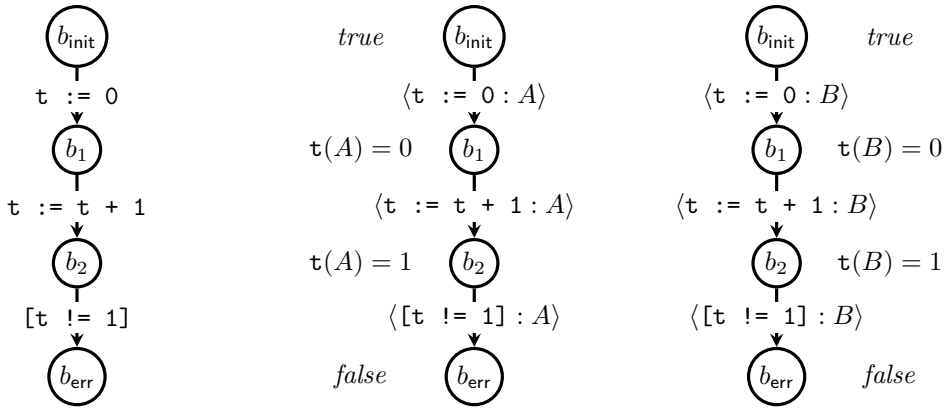
$$\{\Phi^{TM}_{\mathtt{j}}(\ell) \wedge \Phi^{TM}_{\mathtt{i}}(\mathsf{src}(\sigma))\}\ \langle \sigma : \mathtt{i} \rangle\ \{\Phi^{TM}_{\mathtt{j}}(\ell)\}$$

  is valid
- Safety: *for all* $\mathtt{i} \in N$, $\Phi^{TM}_{\mathtt{i}}(\ell_{err}) = \mathit{false}$

Example 4.4.4. *Figure 4.5a depicts a program that is executed by two threads, both of which set a local variable* `t` *to 0, increment* `t`*, and then go to an error location* b_{err} *if* `t` *is not one. Each thread operates on disjoint memory, making it particularly easy to reason about each thread independently. For each thread* `i`*,* `t(i)` *is 0 at* b_1 *and 1 at* b_2*, so the guard on the transition to* b_{err} *is never satisfied. Figure 4.5b depicts a thread-modular proof to this effect.*

Thread-modular proofs are *incomplete* in the sense that there are programs that have Ashcroft/Manna proofs but that do not have thread-modular proofs.[1] The program in Figure 4.4a is a classical example of this phenomenon. Our interest in thread-modular proofs is due to the fact that (when they exist) they are *small*. A thread-modular annotation defines an assertion for each control location of each thread, of which there are $|\mathsf{Loc}| \cdot |N|$.

[1] Completeness can be recovered by introducing *auxiliary variables* to track additional state, an idea due to [Owicki and Gries, 1976]. The Owicki-Gries proof system is equivalent to thread-modular proofs plus a rule for introducing such auxiliary variables.



(a) Thread template (b) Thread-modular proof

Figure 4.5: A thread-modular proof that b_{err} is unreachable in the program where two threads execute the thread template (a). The thread modular proof consists of two annotated copies of the thread template, one for each thread, with the annotation at each control location appearing beside it.

Any thread-modular proof Φ^{TM} can be "compiled" into an Ashcroft/Manna proof Φ^{AM} by defining $\Phi^{AM}(loc) \triangleq \bigwedge_{\mathtt{i} \in N} \Phi^{TM}_{\mathtt{i}}(loc(\mathtt{i}))$ for each control state *loc*. For example, Figure 4.6 shows the result of compiling the thread-modular proof in Figure 4.5b into an Ashcroft/Manna proof following this procedure.

Observe that the examples in Figure 4.4b and Figure 4.6 have Ashcroft/Manna proofs with some redundant structure: there are a small number of relevant assertions, and they are repeated throughout the proof. *Localized proofs* expose this redundancy. The size of a localized proof is taken to be the number of assertions that appear in it. Each control state is labelled by a *set* of assertions that must hold at this point, which allows localized proofs to avoid "double counting" assertions like $\mathtt{t}(A) = 0$ in Figure 4.6, which appear multiple times conjoined with different assertions.

Definition 4.4.5 (k-bounded localized proof)**.** *Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N\rangle$ be a program with a finite set of threads N, let ℓ_{err} be an error location, and let $k \geq 1$. A* k-bounded localized proof *is a map $\Phi : \mathsf{ControlState} \to 2^{\mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}(N))}$ from control locations of P to sets of assertions over the globals and locals of every thread, such that the following conditions hold:*

1. Initiation: $\Phi(loc_{init}) = \emptyset$.

2. Consecution: *For every command $\sigma \in \Sigma$, every thread $\mathtt{i} \in N$, every control state $loc \in \mathsf{ControlState}$ such that $loc(\mathtt{i}) = \mathsf{src}(\sigma)$, and every assertion $\varphi \in \Phi(loc[\mathtt{i} \leftarrow \mathsf{tgt}(\sigma)])$, there exists a subset $\Psi \subseteq \Phi(loc)$ of cardinality $\leq k$ such that the Hoare triple $\{\bigwedge \Psi\}\ \langle \sigma : \mathtt{i}\rangle\ \{\varphi\}$ is valid.*

3. Safety: *For every control state loc and every thread $\mathtt{i}$ such that $loc(\mathtt{i}) = \ell_{err}$, we have $false \in \Phi(loc)$.*

We define the size $size(\Phi) \triangleq |\bigcup_{loc \in \mathsf{ControlState}} \Phi(loc)|$ of a localized safety proof to be the number of assertions it contains.

An Ashcroft/Manna proof corresponds to a 1-bounded localized safety proof, where every location is labelled with a single assertion (except loc_{init}, which is labelled with $\emptyset$). A thread-modular proof Φ^{TM} corresponds to a 2-bounded localized proof, by defining for each $loc \in \mathsf{ControlState}$, $\Phi(loc) \triangleq$

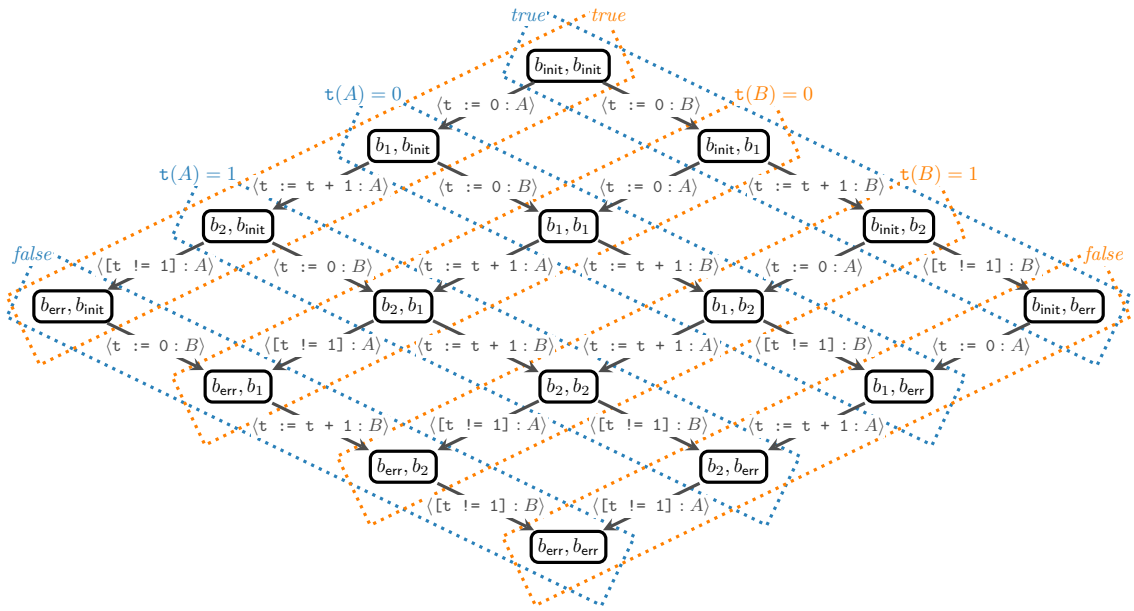


Figure 4.6: Ashcroft/Manna proof corresponding to the thread modular proof Figure 4.5b. Each node is associated with the conjunction of two assertions corresponding to the dotted boxes containing it.

$\{\Phi_{\mathtt{i}}^{TM}(loc(\mathtt{i})) : \mathtt{i} \in N\}$. Thus the size of a localized proof constructed from a thread-modular one is at most $|\mathsf{Loc}| \cdot |N|$.

Finally, we may state the main result of this section. The theorem states that if there is a small proof that a non-reachability property holds (a k-bounded localized proof for some small fixed k that uses few assertions), then there is a small iDFG proof.

Theorem 4.4.6 (Succinctness)**.** *Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N\rangle$ be a program and let $\ell_{err} \in \mathsf{Loc}$ be an error location. Suppose Φ is a k-bounded localized proof that ℓ_{err} is unreachable in P. There exists an iDFG with at most $O(size(\Phi)^{k+1} \cdot |\Sigma| \cdot |N|)$ vertices that proves ℓ_{err} is unreachable in P.*

Proof. Define an iDFG $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathsf{post}}\rangle$ as follows:

- $V \triangleq \{\langle \Psi, \langle \sigma : \mathtt{i}\rangle\rangle : \exists loc \in \mathsf{ControlState}. loc(\mathtt{i}) = \mathsf{src}(\sigma) \wedge \Psi \subseteq \Phi(loc) \wedge |\Psi| \leq k\}$
  That is, each vertex is a command $\langle \sigma : \mathtt{i}\rangle$ paired with a set of preconditions of cardinality $\leq k$.
- $E \triangleq \{\langle \Psi, \langle \sigma : \mathtt{i}\rangle\rangle \xrightarrow{\varphi} \langle \Psi', \langle \sigma' : \mathtt{i}'\rangle\rangle : \{\bigwedge \Psi\}\ \langle \sigma : \mathtt{i}\rangle\ \{\varphi\}$ is valid and $\varphi \in \Psi'\}$
- $\mathsf{cmd}(\Psi, \langle \sigma : \mathtt{i}\rangle) \triangleq \langle \sigma : \mathtt{i}\rangle$
- $V_{\mathsf{final}} \triangleq \{\langle \Psi, \langle \sigma : \mathtt{i}\rangle\rangle \in V : \{\bigwedge \Psi\}\ \langle \sigma : \mathtt{i}\rangle\ \{\mathit{false}\}$ is valid$\}$

Observe that the consecution condition for localized proofs implies that for all vertices $\langle \Psi, \langle \sigma : \mathtt{i}\rangle\rangle \in V$, we have $\mathsf{in}(\Psi, \langle \sigma : \mathtt{i}\rangle) = \Psi$. The inductiveness condition for iDFGs then follows trivially from the construction of E and V_{final}. Clearly $|V|$ is $O(size(\Phi)^{k+1} \cdot |\Sigma| \cdot |N|)$. It remains only to show that $\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(G)$. First, we need a lemma:

Lemma 4.4.7. *Let τ be a non-empty program trace, let $loc \triangleq \lambda \mathtt{j}.\mathit{tgt}(\tau, \mathtt{j})$ be the control state of the program after executing τ, and let $loc' \triangleq loc'[\mathtt{i} \mapsto \mathit{tgt}(\sigma)]$ be the control state of the program after*

executing $\tau\langle\sigma : \mathtt{i}\rangle$. *Let* $\varphi \in \Phi(loc')$, *and let* $\Psi \subseteq \Phi(loc)$ *be a subset of cardinality* $\leq k$ *such that* $\{\bigwedge \Psi\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi\}$ *is valid. Then* $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}_G(\langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle, \varphi)$.

Before we prove the lemma, we will use it to prove $\mathcal{L}(P, \ell_{\mathsf{err}}) \subseteq \mathcal{L}(G)$. First, observe that every trace in $\mathcal{L}(P, \ell_{\mathsf{err}})$ is a non-empty program trace of the form $\tau\langle\sigma : \mathtt{i}\rangle$. Let $loc \triangleq \lambda \mathtt{j}.\mathsf{tgt}(\tau, \mathtt{j})$ be the control state of the program after executing τ, and let $loc' \triangleq loc'[\mathtt{i} \mapsto \mathsf{tgt}(\sigma)]$ be the control state of the program after executing $\tau\langle\sigma : \mathtt{i}\rangle$. By the safety condition of localized proofs (and the fact that $\tau\langle\sigma : \mathtt{i}\rangle$ is an error trace), we have $\mathit{false} \in \Phi(loc')$. By the consecution condition of localized proofs, there is some $\Psi \subseteq \Phi(loc)$ of cardinality $\leq k$ such that $\{\bigwedge \Psi\}\ \langle\sigma : \mathtt{i}\rangle\ \{\mathit{false}\}$ is valid, and therefore $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}_G(\langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle, \mathit{false})$ by the lemma. Finally, $\langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle \in V_{\mathsf{final}}$ by the definition of V_{final}, so that $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}_G(\langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle, \mathit{false}))$ implies $\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}(G)$ by the definition of $\mathcal{L}(G)$.

We now prove the lemma. Let $\tau, \sigma, \mathtt{i}, \varphi, \Psi, loc, loc'$ be as in the statement of the lemma. We prove (cf. case 2 of Definition 4.2.2) that

(*) for all $\varphi \in \mathsf{in}(\Psi, \langle\sigma : \mathtt{i}\rangle)$ there exists some $u \xrightarrow{\varphi} \langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle$ such that $\tau \in \mathcal{L}_G(u, \varphi)$.

We proceed by induction on the length of traces.

- Base case $\tau = \epsilon$. By definition, we have $loc = loc_{\mathsf{init}}$. By the initiation condition, we must have $\Phi(loc) = \emptyset$, and since $\Psi \subseteq \Phi(loc)$ we have $\Psi = \emptyset$. By construction of E, $\mathsf{in}(\Psi, \langle\sigma : \mathtt{i}\rangle) = \emptyset$, and (*) holds vacuously.

- Induction step $\tau = \tau'\langle\sigma' : \mathtt{i}'\rangle$. Towards (*), let $\varphi \in \mathsf{in}(\Psi, \langle\sigma : \mathtt{i}\rangle)$. Let $loc'' \triangleq loc[\mathtt{i}' \leftarrow \mathsf{src}(\sigma')]$. By the consecution condition for localized proofs, there is a set $\Psi' \subseteq \Phi(loc'')$ of cardinality $\leq k$ such that the Hoare triple $\{\bigwedge \Psi'\}\ \langle\sigma' : \mathtt{i}'\rangle\ \{\varphi\}$ is valid. By the construction of E, we have $\langle\Psi', \langle\sigma' : \mathtt{i}'\rangle\rangle \xrightarrow{\varphi} \langle\Psi, \langle\sigma : \mathtt{i}\rangle\rangle \in E$. By the induction hypothesis, we have $\tau'\langle\sigma' : \mathtt{i}'\rangle \in \mathcal{L}_G(\langle\Psi', \langle\sigma' : \mathtt{i}'\rangle\rangle, \varphi)$, and we are done. □

4.5 Verification Algorithm

This section describes an iDFG-based algorithm for verifying and refuting reachability properties of concurrent programs. Given a program P and an error location ℓ_{err}, the goal of this algorithm is to either:

- construct an iDFG that proves that ℓ_{err} is unreachable in P, or
- construct a *counter-example* proving that ℓ_{err} is reachable (that is, a feasible error trace).

The algorithm follows the outline of Figure 3.6 (where naturally, the proof objects *IP* in the diagram are instantiated to iDFGs). The algorithm builds an iDFG by repeatedly sampling error traces, constructing iDFG proofs of their infeasibility, and then assembling the iDFGs into an iDFG that recognizes every error trace of the program. Section 4.3 shows that we can implement the proof checking step of this algorithm using alternating finite automata. The remaining questions to be answered are (1) how to construct an iDFG proof for a single trace, and (2) how to *combine* ("*assemble*") two iDFGs into a single iDFG that recognizes more traces than either of its components. Section 4.5.1 answers the first question, and Section 4.5.2 the second. Finally, Section 4.5.3 states the software model checking algorithm explicitly.

4.5.1 Constructing an iDFG from a trace

This section describes an algorithm construct-idfg that constructs an iDFG proof for a single trace. Given a trace τ and a post-condition φ such that $\{true\}\ \tau\ \{\varphi\}$ is valid, construct-idfg(τ, φ) computes an iDFG with post-condition φ that recognizes τ. By Proposition 4.2.3, such an iDFG serves as a proof that $\{true\}\ \tau\ \{\varphi\}$ is a valid Hoare triple.

The algorithm uses an auxiliary procedure Interpolate to synthesize proof-relevant assertions. Given a trace τ, a command $\langle \sigma : \mathtt{i} \rangle$, and an assertion φ such that the Hoare triple $\{true\}\ \tau\langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid, Interpolate$(\tau, \langle \sigma : \mathtt{i} \rangle, \varphi)$ computes a set of *intermediate assertions* Ψ such that both

$$\{true\}\ \tau'\ \{\bigwedge \Psi\} \quad \text{and} \quad \{\bigwedge \Psi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$$

are valid Hoare triples. Without loss of generality, we suppose Interpolate returns a set that does *not* contain any tautologies. Interpolate can be implemented using a variety of techniques, including Craig interpolation [Henzinger et al., 2004b, Craig, 1957], predicate transformers [Dijkstra, 1975], or abstract transformers in an abstract domain [Cousot and Cousot, 1977]. The division of the intermediate assertion into a set of assertions can be accomplished simply by splitting conjunctions, or more sophisticated heuristics may be developed.

Input : Non-empty trace τ and an assertion φ such that $\{true\}\ \tau\ \{\varphi\}$ is valid
Output: An iDFG G_τ with post-condition φ and such that $\tau \in \mathcal{L}(G_\tau)$)
Let $\tau'\langle \sigma : \mathtt{i} \rangle = \tau$;
$\Psi \leftarrow$ Interpolate$(\tau', \langle \sigma : \mathtt{i} \rangle, \varphi')$;
if $\Psi = \{\varphi\}$ **then**
  /* $\langle \sigma : \mathtt{i} \rangle$ is irrelevant */
  return construct-idfg(τ', φ')
else
  $v \leftarrow$ fresh vertex;
  cmd$(v) \leftarrow \langle \sigma : \mathtt{i} \rangle$;
  $V \leftarrow \{v\}$;
  $E \leftarrow \emptyset$;
  for $\psi \in \Psi$ **do**
    $\langle V_\psi, E_\psi, \mathsf{cmd}_\psi, V^i_{\mathsf{final}}, \psi \rangle \leftarrow$ construct-idfg(τ', ψ);
    $V \leftarrow V \cup V_\psi$;
    $E \leftarrow E \cup E_\psi \cup \{u \xrightarrow{\psi} v : u \in V^i_{\mathsf{final}}\}$;
    for $v \in V_\psi$ **do**
      cmd$(v) \leftarrow \mathsf{cmd}_\psi(v)$;
    end
  end
  return $G_\tau = \langle V, E, \mathit{cmd}, \{v\}, \varphi \rangle$
end

Algorithm 3: construct-idfg(τ, φ)

Intuitively, the procedure construct-idfg(τ, φ) propagates the post-condition φ backwards across τ; along the way it detects the commands and ordering constraints in τ that are relevant for attaining the post-condition φ. Detecting (ir)relevant commands and ordering constraints leverages the Interpolate procedure as follows:

- If Interpolate$(\tau', \langle \sigma : \mathtt{i} \rangle, \varphi)$ returns the singleton $\{\varphi\}$ containing the post-condition φ, then the

command $\langle \sigma : \mathtt{i} \rangle$ is irrelevant to achieving φ. From the specification of Interpolate, we know that φ is stable under $\langle \sigma : \mathtt{i} \rangle$ (i.e., $\{\varphi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid), so $\langle \sigma : \mathtt{i} \rangle$ may be suppressed in the iDFG.

- If $\mathsf{Interpolate}(\tau', \langle \sigma : \mathtt{i} \rangle, \varphi)$ returns a set Ψ (with $\Psi \neq \{\psi\}$), then τ' must have each $\psi \in \Psi'$ as a post-condition. The order in which these conditions are achieved is irrelevant. The iDFG construction constructs one iDFG for each $\psi \in \Psi'$ independently. In an efficient implementation, this step may happen in parallel.

Note that construct-idfg produces iDFGs of a very particular form: the graphs are acyclic, and every vertex has exactly one incoming edge for each assertion (i.e., there are no disjunctive choices). The *merge* procedure introduced in the next section is responsible for adding cycles and disjunctive choices to iDFGs.

4.5.2 Merging iDFGs

Suppose that we have two iDFGs G_1 and G_2, each of which recognizes some subset of the error traces of the program. This section is concerned with the question of how we can combine G_1 and G_2 into an iDFG that recognizes all the traces of G_1 and G_2 (and typically, even more). The merge operator $G_1 \between G_2$ described in this section can be thought of as a three step process: in the first step, we construct the disjoint union of G_1 and G_2; in the second step, *completion*, we saturate this iDFG by adding edges that do not violate the inductiveness property of iDFGs; in the third step, *reduction*, we collapse "equivalent" vertices.

First, we observe that iDFGs with the same post-condition can be combined via union. The disjoint union $G_1 \cup G_2$ recognizes exactly the set of traces that are recognized by G_1 or G_2. The operation is defined explicitly as:

Definition 4.5.1 (Union)**.** *Let $G_1 = \langle V_1, E_1, \mathsf{cmd}_1, V^1_{\mathsf{final}}, \varphi_{post} \rangle$ and $G_2 = \langle V_2, E_2, \mathsf{cmd}_2, V^2_{\mathsf{final}}, \varphi_{post} \rangle$ be iDFGs that share the same post-condition φ_{post}. Without loss of generality, suppose V_1 and V_2 are disjoint (if not, we may rename the vertices in V_2). We define the union of G_1 and G_2 as:*

$$G_1 \cup G_2 \triangleq \langle V_1 \cup V_2, E_1 \cup E_2, \mathsf{cmd}^{\cup}, V^1_{\mathsf{final}} \cup V^2_{\mathsf{final}}, \varphi_{post} \rangle$$

where $\mathsf{cmd}^{\cup}(v) \triangleq \begin{cases} \mathsf{cmd}_1(v) & \text{if } v \in V_1 \\ \mathsf{cmd}_2(v) & \text{if } v \in V_2 \end{cases}$.

It is straightforward to prove that $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1 \cup G_2)$ directly. However, we will develop an *embedding lemma*, from which $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1 \cup G_2)$ is a trivial consequence, that will be useful later in the section.

We say that an iDFG G embeds into G' if G can be mapped onto a sub-graph of G' in a way that is, in some sense, "tight." Formally, we define an iDFG embedding as follows:

Definition 4.5.2 (Embedding)**.** *Given inductive data flow graphs $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{post} \rangle$ and $G' = \langle V', E', \mathsf{cmd}', V'_{\mathsf{final}}, \varphi_{post} \rangle$ (with the same post-condition) we say that a map $h : V \to V'$ is an* embedding *if the following hold:*

- *For all vertices $v \in V$, v and $h(v)$ have the same label and inputs ($\mathsf{cmd}(v) = \mathsf{cmd}'(h(v))$ and $\mathsf{in}_G(v) = \mathsf{in}_{G'}(h(v))$).*

- *For all final vertices* $v \in V_{\mathsf{final}}$, $h(v) \in V'_{\mathsf{final}}$.
- *For all edges* $u \xrightarrow{\varphi} v$ *in* E, $h(u) \xrightarrow{\varphi} h(v)$ *is in* E'.

If such an embedding exists, we say that G embeds *into* G'.

The following is the crucial property concerning iDFG embeddings.

Lemma 4.5.3 (Embedding lemma)**.** *If* G *and* G' *are iDFGs such that* G *embeds into* G'*, then* $\mathcal{L}(G) \subseteq \mathcal{L}(G')$.

Proof. Let $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{\mathsf{post}} \rangle$ and $G' = \langle V', E', \mathsf{cmd}', V'_{\mathsf{final}}, \varphi_{\mathsf{post}} \rangle$ be two iDFGs and let h be an embedding of G into G'. We prove that for all $v \in V$ and all formulas φ, $\mathcal{L}_G(v, \varphi) \subseteq \mathcal{L}_{G'}(h(v), \varphi)$. The main result then follows by the following calculation.

$$
\begin{aligned}
\mathcal{L}(G) &= \bigcup_{v \in V_{\mathsf{final}}} \mathcal{L}_G(v, \varphi_{\mathsf{post}}) && \text{By definition of } \mathcal{L}(G) \\
&\subseteq \bigcup_{v \in V_{\mathsf{final}}} \mathcal{L}_{G'}(h(v), \varphi_{\mathsf{post}}) && \text{By above} \\
&\subseteq \bigcup_{v' \in V'_{\mathsf{final}}} \mathcal{L}_{G'}(v', \varphi_{\mathsf{post}}) && h \text{ is an embedding} \\
&= \mathcal{L}(G') && \text{By definition of } \mathcal{L}(G') \ .
\end{aligned}
$$

The proof of the lemma is by induction on the length of τ. The base case is vacuous, because $\epsilon \notin \mathcal{L}_G(v, \varphi)$ by definition. Following Definition 4.2.2, there are two cases to consider for the inductive step:

- Case $\{\varphi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid and $\tau \in \mathcal{L}_G(v, \varphi)$. Then by the induction hypothesis, we have $\tau \in \mathcal{L}_{G'}(h(v), \varphi)$ and the result follows immediately.
- Case $\mathsf{cmd}(v) = \langle \sigma : \mathtt{i} \rangle$ and $\forall \psi \in \mathsf{in}(v). \exists u \xrightarrow{\psi} v \in E. \tau \in \mathcal{L}_G(u, \psi)$. Since h is an embedding, we have

  (i) $\mathsf{cmd}(h(v)) = \langle \sigma : \mathtt{i} \rangle = \mathsf{cmd}(v)$,

  (ii) $\mathsf{in}_G(v) = \mathsf{in}_{G'}(h(v))$, and

  (iii) for all $u \xrightarrow{\psi} v$ in E, $h(u) \xrightarrow{\psi} h(v)$ belongs to E'.

  We have

$$
\begin{aligned}
&\forall \psi \in \mathsf{in}(v). \exists u \xrightarrow{\psi} v \in E. \tau \in \mathcal{L}_G(u, \psi) && \text{Assumption} \\
&\implies \forall \psi \in \mathsf{in}(v). \exists u \xrightarrow{\psi} v \in E. \tau \in \mathcal{L}_{G'}(h(u), \psi) && \text{Induction hypothesis} \\
&\iff \forall \psi \in \mathsf{in}(h(v)). \exists u \xrightarrow{\psi} v \in E. \tau \in \mathcal{L}_{G'}(h(u), \psi) && \text{(ii)} \\
&\implies \forall \psi \in \mathsf{in}(h(v)). \exists u \xrightarrow{\psi} h(v) \in E'. \tau \in \mathcal{L}_{G'}(u, \psi) && \text{(iii)}
\end{aligned}
$$

  Finally, we have $\tau \langle \sigma : \mathtt{i} \rangle \in \mathcal{L}_{G'}(h(v), \psi)$ by the above, (i), and the definition of $\mathcal{L}_{G'}$. □

In the software model checking algorithm described in this section, reduction and completion play the crucial role of introducing cycles into iDFGs. As we mentioned in Section 4.5.1, construct-idfg produces

acyclic iDFGs, which cannot account for non-trivial repetitive behaviour. Reduction introduces cycles into iDFGs by collapsing vertices with the same labels an inputs. Completion introduces cycles into iDFGs by saturating them with all edges that do not violate the consecution condition. A declarative definition for what it means for an iDFG to be reduced and complete is as follows:

Definition 4.5.4 (Reduced, Complete)**.** *Let $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{post}\rangle$ be an iDFG.*

- *We say that G is* complete *if:*
  - *For any $v \in V$ such that $\{\bigwedge \mathsf{in}(v)\}\ \mathsf{cmd}(v)\ \{\varphi_{post}\}$ holds, then $v \in V_{\mathsf{final}}$, and*
  - *For any $u, v \in V$, and any $\varphi \in \mathsf{in}(v)$ such that $\{\bigwedge \mathsf{in}(u)\}\ \mathsf{cmd}(u)\ \{\varphi\}$ is valid, we have $u \xrightarrow{\varphi} v \in E$.*
- *We say that G is* reduced *if there exist no distinct $u, v \in V$ such that $\mathsf{cmd}(u) = \mathsf{cmd}(v)$ and $\mathsf{in}(u) = \mathsf{in}(v)$.*

There is a straight-forward procedure that, given an arbitrary iDFG G, constructs a reduced complete iDFG $rc(G)$ such that $\mathcal{L}(rc(G))$ contains $\mathcal{L}(G)$. We call $rc(G)$ the *reduced completion of* G. This is stated formally in the following proposition:

Proposition 4.5.5. *For every iDFG $G = \langle V, E, \mathsf{cmd}, V_{\mathsf{final}}, \varphi_{post}\rangle$, there is a reduced complete iDFG $rc(G)$ that can be computed from G in $O(|V| \cdot |E|)$ space and such that G and $rc(G)$ have the same post-condition and $\mathcal{L}(G) \subseteq \mathcal{L}(rc(G))$.*

Proof. We define $rc(G) \triangleq \langle V', E', \mathsf{cmd}', V'_{\mathsf{final}}, \varphi_{\mathsf{post}}\rangle$, where

$$
\begin{aligned}
V' &\triangleq \{\langle \mathsf{in}_G(v), \mathsf{cmd}(v)\rangle : v \in V\} \\
E' &\triangleq \{\langle \Psi, \langle \sigma : \mathtt{i}\rangle\rangle \xrightarrow{\varphi} \langle \Psi', \langle \sigma' : \mathtt{j}\rangle\rangle : \varphi \in \Psi' \wedge \{\textstyle\bigwedge \Psi\}\ \langle \sigma : \mathtt{i}\rangle\ \{\varphi\} \text{ is valid}\} \\
\mathsf{cmd}'(\langle \sigma : \mathtt{i}\rangle, \Psi) &\triangleq \langle \sigma : \mathtt{i}\rangle \\
V'_{\mathsf{final}} &\triangleq \{(\Psi, \langle \sigma : \mathtt{i}\rangle) : \{\textstyle\bigwedge \Psi\}\ \langle \sigma : \mathtt{i}\rangle\ \{\mathit{false}\} \text{ is valid}\}
\end{aligned}
$$

It is easy to check that $rc(G)$ is well-defined. Clearly, G embeds into $rc(G)$, using the map that sends every vertex v to $\langle \mathsf{in}_G(v), \mathsf{cmd}(v)\rangle$. The fact that $\mathcal{L}(G) \subseteq \mathcal{L}(rc(G))$ then follows from the embedding lemma. □

Finally, we describe the merge operation. The merge operator $G_1 \between G_2$ functions by forming the disjoint union of G_1 and G_2, and then taking the reduced completion of the resulting iDFG. Formally, we define merge as follows:

Definition 4.5.6. *Given two disjoint iDFGs sharing the same post-condition $G_1 = \langle V_1, E_1, \mathsf{cmd}_1, V^1_{\mathsf{final}}, \varphi_{post}\rangle$ and $G_2 = \langle V_2, E_2, \mathsf{cmd}_2, V^2_{\mathsf{final}}, \varphi_{post}\rangle$, their* merge *$G_1 \between G_2$ is defined to be $rc(G_1 \cup G_2)$.*

Lemma 4.5.7. *Let G_1 and G_2 be disjoint iDFGs with the same post-condition. We have*

$$\mathcal{L}(G_1) \cup \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1 \between G_2).$$

Proof. First, note that G_1 and G_2 both embed into $G_1 \cup G_2$, so $\mathcal{L}(G_1) \cup \mathcal{L}(G_2)$ follows from the embedding lemma. Then we have $\mathcal{L}(G_1) \cup \mathcal{L}(G_2) \subseteq \mathcal{L}(G_1 \between G_2) = \mathcal{L}(rc(G_1 \cup G_2))$ by Proposition 4.5.5. □

4.5.3 Putting it all together

Finally, we collect all the preceding results into a software model checking algorithm, following the trace abstraction paradigm. The algorithm appears in Algorithm 4.

Input : Program P, error location ℓ_{err}
Output: Safe if ℓ_{err} is unreachable; feasible error trace otherwise.
`/* G is initialized to an empty iDFG with post-condition` *false* `*/`
$\mathsf{cmd} \leftarrow$ empty map;
$G \leftarrow \langle \emptyset, \emptyset, \mathsf{cmd}, \emptyset, \mathit{false} \rangle$;
while $\mathcal{L}(P, \ell_{\mathsf{err}}) \not\subseteq \mathcal{L}(G)$ **do**
  Pick error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}}) \setminus \mathcal{L}(G)$;
  if τ *is feasible* **then**
    return *Unsafe: counter-example* τ
  else
    $G_\tau \leftarrow \mathsf{construct\text{-}idfg}(\tau, \mathit{false})$;
    $G \leftarrow G \between G_\tau$
  end
end
return *Safe*

Algorithm 4: Software model checking using iDFGs

Partial correctness of Algorithm 4 is stated as follows:

Theorem 4.5.8 (Partial Correctness)**.** *Let P be a program and let ℓ_{err} be an error location. If Algorithm 4 returns Safe given P and ℓ_{err} as inputs, then ℓ_{err} is unreachable in P. If Algorithm 4 returns Unsafe, then ℓ_{err} is reachable.*

Also note that Algorithm 4 is complete for refutation, as long as the step "Pick new $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}}) \setminus \mathcal{L}(G)$" chooses a *shortest* trace. Choosing a shortest trace ensures that the algorithm eventually enumerates all error traces, so that in particular it will enumerate a feasible error trace if one exists.

4.6 Related work

Finite-state model checking. One of the classical strengths of model checking is in verifying and refuting temporal properties of finite-state concurrent systems [Clarke et al., 1983]. Much of this work is centred around making exhaustive search of the state space of concurrent systems more efficient. Within this line of work, partial order reduction is the technique perhaps the most relevant to this chapter [Valmari, 1991, Peled, 1993, Godefroid, 1994]. Partial order reduction exploits the observation that in a concurrent system some commands may commute (in the sense that the end state is not affected by the order in which the commands are executed), making some interleavings redundant. Such redundant interleavings can be eliminated from the model while preserving a class of properties of interest. Like iDFGs, partial order reduction attempts to avoid the interleaving model of concurrency in favour of a partially-ordered model [Mazurkiewicz, 1986]. An interesting contrast between techniques based on partial order reduction and iDFGs is that partial order reduction exploits parallelism in programs, but iDFGs exploit parallelism in *proofs*. For example, consider the iDFG in Figure 4.1b. The commands $\langle$`x := 0 : 1`$\rangle$ and $\langle$`x := 1 : 2`$\rangle$ do not commute since they write to the same variable; however, their order does not matter for the proof. Figure 4.2 pictures three traces, *none* of which are equivalent up to commuting independent events, and yet have the same iDFG proof.

Symmetry reduction is a model-checking technique that avoids exploring redundant behaviours in concurrent systems with identical processes [Emerson and Sistla, 1996]. This sort of symmetry is not exploited by iDFGs, but we will make essential use of it in the next chapter on proof spaces.

Finite state model checking is not directly relevant to program verification because programs have infinitely many states, but can be applied indirectly by first extracting a finite abstraction of an infinite state program using predicate abstraction [Graf and Saïdi, 1997] and verifying properties of the abstraction (the so-called *reductionist* approach to model checking software). The process of model-checking finite abstractions of infinite-state programs is sound for verification (a property that holds in the abstraction holds in the concrete program) but not refutation (an error in the abstraction does not necessarily correspond to an error in the program). However, a spurious counter-example that refutes of a property in the abstraction can be used to refine it [Clarke et al., 2000]. There exist even specialized techniques for generating good refinements in a concurrent setting [Donaldson et al., 2011]. This process can be used iteratively, constructing increasingly precise abstractions until either a true counter-example is found or an abstraction that is precise enough to verify the property of interest. This algorithm is known as counter-example guided abstraction refinement (CEGAR).

Trace abstraction [Heizmann et al., 2009], which underlies the iDFG software model checking procedure Algorithm 4, serves an intuitively similar role to CEGAR. The role of abstraction is different in the two approaches. In CEGAR, a coarse model (containing many behaviours not necessarily corresponding to executions of the program) is iteratively refined by removing behaviours until all behaviours of the model are correct. In trace abstraction, a proof object is iteratively augmented by adding correct behaviours until all behaviours of the program are contained in the proof object. One may consolidate the two approaches by viewing the *complement* of a trace abstraction as defining an abstract model (albeit, not of the typical sort obtained via predicate abstraction). In this sense, we may conceive of an iDFG as a description of a particular type of finite-state abstraction, where the finite model is the state space of the alternating finite automaton obtained by intersecting the error trace automaton $\mathcal{A}(P, \ell_{\mathsf{err}})$ with the complement of the automaton $\mathcal{A}(G)$ (for some program P, error location ℓ_{err}, and iDFG G). Finite-state model checking is complementary to the approach described in this chapter in the sense that the various techniques that have been developed for exhaustive exploration of finite state spaces (partial order reduction, symmetry reduction, symbolic techniques) may be exploited for verifying that all traces of a program are recognized by an iDFG.

Lazy abstraction. In the world of sequential program verification, the reductionist approach has been surpassed in performance by *lazy* abstraction techniques [Henzinger et al., 2002, McMillan, 2006]. The idea behind lazy abstraction is to tightly integrate abstraction, model checking, and refinement rather than separate them into distinct phases (as in CEGAR). Instead of constructing a sequence of increasingly precise abstract models, lazy abstraction constructs a single abstract model with variable precision at different points in the model. This variable precision overcomes a limitation of classical predicate abstraction, where the abstraction is defined by a single set of predicates: it is computationally expensive to compute the transition relation of a model with a large set of predicates, and using only *relevant* predicates at different points in the model reduces the expense. McMillan's Impact algorithm takes this idea to the extreme and avoids computing the transition relation entirely, instead relying entirely on refinement to drive the synthesis of the abstract state space [McMillan, 2006]. Trace abstraction, and iDFGs in particular, can be seen as a hybrid between CEGAR and Impact. Like CEGAR, trace

abstraction separates abstraction, model checking and refinement phases; but like Impact, the abstraction phase avoids computing transition abstractions, and has varying degrees of precision throughout the proof.

The tight integration of abstraction, model checking, and refinement complicates the combination of lazy abstraction and state space reduction techniques, but some techniques have been proposed. [Cimatti et al., 2011] combines lazy abstraction with partial order reduction for verifying SystemC programs. [Wachter et al., 2013] integrates Impact with dynamic partial order reduction. [Brückner et al., 2007, Dräger et al., 2010] combines lazy abstraction and *slicing* to reduce the state space.

Thread-modular reasoning The tools of software model checking have also found use in automatic synthesis of thread-modular proofs. [Gupta et al., 2011] presents a technique for automated Rely-Guarantee reasoning [Jones, 1981] using predicate abstraction. Constraints for generating predicates to refine abstractions are encoded into recursion-free Horn clauses, allowing a great deal of flexibility in terms of encoding preferences (such as *prefer predicates that do not relate local variables of different threads*). Using recursion-free Horn clauses to construct an iDFG from a trace is a promising direction for future research. [Grebenshchikov et al., 2012] shows that many different program logics, including Owicki-Gries [Owicki and Gries, 1976] and Rely-Guarantee [Jones, 1981] can be encoded as recursive Horn constraints, thus enabling off-the-shelf Horn clause solvers to automatically synthesize proofs in these logics.

Thread-modular model checking [Flanagan et al., 2002] can be seen as a method for generating thread-modular proofs (in the style of Definition 4.4.3) for finite state concurrent programs. The approach of thread-modular model checking can be brought to the domain of infinite-state programs using a CEGAR approach [Henzinger et al., 2003].

Chapter 5

Proof Spaces

This chapter introduces *proof spaces*, a proof system for reasoning about concurrent programs with an unbounded number of threads. Proof spaces can be seen as a deconstruction of the principles behind inductive data flow graphs, generalized to handle an arbitrary number of threads. Departing from the graphical models in the previous two chapters, proof spaces are abstract mathematical structures: a proof space is simply a collection of Hoare triples closed under certain rules of inference. The inference rules allow a large set of theorems to be derived from a small set of axiomatic triples, enabling succinct proofs of multi-threaded programs. Like iDFGs, proof spaces can be used as a basis for software model checking following the trace abstraction paradigm [Heizmann et al., 2009].

5.1 Overview

We start by demonstrating proof spaces on a simple example. Consider the program in which an infinite number of threads (identified by natural numbers) execute the code below in parallel. The goal is to verify that, if $\mathtt{g} \geq 1$ holds initially, then it will always hold (regardless of how many threads are executing).

```
global int g
local  int x
1:  x := g;
2:  g := g + x;
```

Consider the set of the Hoare triples (A) - (D) given below.

$$\begin{array}{lrll}
\text{(A)} & \{\mathtt{g} \geq 1\} & \langle \mathtt{x} := \mathtt{g} : 1 \rangle & \{\mathtt{x}(1) \geq 1\} \\
\text{(B)} & \{\mathtt{g} \geq 1 \wedge \mathtt{x}(1) \geq 1\} & \langle \mathtt{g} := \mathtt{g} + \mathtt{x} : 1 \rangle & \{\mathtt{g} \geq 1\} \\
\text{(C)} & \{\mathtt{g} \geq 1\} & \langle \mathtt{x} := \mathtt{g} : 1 \rangle & \{\mathtt{g} \geq 1\} \\
\text{(D)} & \{\mathtt{x}(1) \geq 1\} & \langle \mathtt{x} := \mathtt{g} : 2 \rangle & \{\mathtt{x}(1) \geq 1\}
\end{array}$$

Consider a proof calculus in which these triples are taken as axioms, and the rules of inference are Sequencing, Symmetry, and Conjunction. These rules are easily illustrated with concrete examples:

- Sequencing composes two Hoare triples such that the post-condition of the first implies the

pre-condition of the second. For example, sequencing (A) and (D) yields

$$(\text{A} \,;\, \text{D}) \quad \{\mathsf{g} \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 1 \rangle \langle \mathsf{x} := \mathsf{g} : 2 \rangle \quad \{\mathsf{x}(1) \geq 1\} \,.$$

- SYMMETRY permutes thread identifiers. For example, renaming (A) and (C) (mapping thread $1 \mapsto 2$) yields

$$(\text{A'}) \quad \{\mathsf{g} \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 2 \rangle \quad \{\mathsf{x}(2) \geq 1\}$$
$$(\text{C'}) \quad \{\mathsf{g} \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 2 \rangle \quad \{\mathsf{g} \geq 1\}$$

  and renaming (D) (mapping $1 \mapsto 2$ and $2 \mapsto 1$) yields

$$(\text{D'}) \quad \{\mathsf{x}(2) \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 1 \rangle \quad \{\mathsf{x}(2) \geq 1\} \,.$$

- CONJUNCTION composes two Hoare triples by conjoining pre- and post-conditions. For example, conjoining (A') and (C') yields

$$(\text{A'} \wedge \text{C'}) \quad \{\mathsf{g} \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 2 \rangle \quad \{\mathsf{g} \geq 1 \wedge \mathsf{x}(2) \geq 1\} \,,$$

  and conjoining (A) and (D') yields

$$(\text{A} \wedge \text{D'}) \quad \{\mathsf{g} \geq 1 \wedge \mathsf{x}(2) \geq 1\} \quad \langle \mathsf{x} := \mathsf{g} : 1 \rangle \quad \{\mathsf{x}(1) \geq 1 \wedge \mathsf{x}(2) \geq 1\} \,.$$

A *proof space* is a set of valid Hoare triples that is closed under SEQUENCING, SYMMETRY, and CONJUNCTION (that is, it is a *theory* of this proof calculus). Any finite set of valid Hoare triples generates an infinite proof space by considering those triples to be axioms and taking their closure under the inference rules; we call such a finite set of Hoare triples a *basis* for the generated proof space.

Note that sequencing and conjunction are precisely the rules that justify the correctness argument of iDFGs (Proposition 4.2.3). Indeed, a reduced complete iDFG (Definition 4.5.4) can be thought of as a variation of a proof space that omits the necessity of closure under the symmetry rule. Adding the symmetry rule allows the framework to be applied to programs with an arbitrary number of threads.

Interestingly, the Hoare triples (A) – (D) along with the inference rules (SEQUENCING, SYMMETRY, and CONJUNCTION) are sufficient to prove the correctness of *any* trace of the example program. That is, for any trace τ of the program, $\{\mathsf{g} \geq 1\}\ \tau\ \{\mathsf{g} \geq 1\}$ belongs to the proof space generated by (A) – (D), regardless of *which* or *how many* threads execute in τ. This raises an important algorithmic question of *how can we mechanically check this fact?* As we argued previously, the distinction between formal proof systems and ad-hoc reasoning is that proofs in formal systems can be checked mechanically. To answer this question, we devise *predicate automata*, a class of infinite-state automata that accept languages over an infinite alphabet. We show that, given a program P, an error location ℓ_{err}, and finite basis H for a proof space, the question of whether every error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$ is proved infeasible by the proof space ($\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$ is derivable from H using SEQUENCING, SYMMETRY, and CONJUNCTION) can be reduced to the emptiness problem for predicate automata. We prove that emptiness checking is undecidable for general predicate automata, but we give a semi-algorithm that is sound and that is complete for non-emptiness. We show that the semi-algorithm is in fact a decision procedure for a class of predicate automaton that corresponds to proof spaces where each assertion refers to the local variables

of at most one thread. We introduce *emptiness certificates*, a formal system that can be used to prove emptiness of predicate automata in the case that the semi-algorithm fails to terminate, and which can be mechanically checked.

To appreciate the simplicity proof spaces, consider the following inductive invariant for the program, a classical notion of correctness proof for multi-threaded programs [Ashcroft, 1975]:

$$\mathtt{g} \geq 1 \wedge (\forall i \in \mathtt{Thread}.\mathit{loc}(i) = 2 \Rightarrow \mathtt{x}(i) \geq 1) \ .$$

The invariant indicates that g is at least 1, and all threads i at line 2 of the program have $\mathtt{x}(i)$ at least 1. This invariant is fairly simple, but it makes use of features that are exotic from the standpoint of sequential program verification: thread quantification ($\forall i \in \mathtt{Thread}$), and control predicates ($\mathit{loc}(i) = 2$). In contrast, the triples (A) – (D) are of the form one might expect to generate from a trace using a sequential verifier. Indeed, in Section 5.5 we will show how proof spaces can be used to leverage technology from sequential verification for verifying multi-threaded programs following the trace abstraction paradigm [Heizmann et al., 2009]. Phrased differently: rather than enriching the language of assertions (with thread quantification and control predicates) and employing powerful symbolic reasoning, proof spaces use simple assertions that can be combined using combinatorial reasoning (i.e., without a theorem prover). We show that despite restricting proof spaces to a simple assertional language, they are complete relative to the inductive invariant method in which both thread quantification and control predicates are admitted into the language of assertions.

The organization of this chapter is as follows:

- Section 5.2 defines proof spaces and formulates their associated proof rule for verifying non-reachability properties.
- Section 5.3 shows how the premise of the proof space rule can be checked mechanically using *predicate automata*.
- Section 5.4 investigates the expressive power of proof spaces. We prove that proof spaces are complete with respect the inductive invariant method [Ashcroft, 1975].
- Section 5.5 shows how proof spaces can be used in software model checking.
- Section 5.6 compares proof spaces with related work.

5.2 Proof spaces

We will now introduce *proof spaces*, the central technical idea of this chapter. A proof space is simply a set of Hoare triples that is closed under the inference rules of Sequencing, Symmetry, and Conjunction.

Given the prominence of conjunction in proof spaces, it is convenient to use a slight variation of Hoare triples in which the pre-condition and post-condition are *sets* of assertions, interpreted conjunctively. For the remainder of this chapter a Hoare triple is of the form $\{\varphi_1 \wedge \cdots \wedge \varphi_m\}\ \langle \sigma : \mathtt{i} \rangle\ \{\psi_1 \wedge \cdots \wedge \psi_n\}$ where $\varphi_1 \wedge \cdots \wedge \varphi_m$ and $\psi_1 \wedge ... \wedge \psi_n$ denote the sets $\{\varphi_1, ..., \varphi_m\}$ and $\{\psi_1, ..., \psi_n\}$, respectively. We identify the assertion *true* with the empty conjunction.

The *sequencing rule* is a variant of the familiar one from Hoare logic. It allows two Hoare triples to be composed sequentially to prove a property of longer trace (and in particular, repeated applications

of the sequencing rule allow proof spaces to reason about loops). We omit the rule of consequence from Hoare logic, and instead incorporate consequence in sequencing: we allow triples $\{\varphi_0\}\ \tau_0\ \{\varphi_1\}$ and $\{\varphi_1'\}\ \tau_1\ \{\varphi_2\}$ to be composed when φ_1 entails φ_1'. A design goal of proof spaces is that the inference rules should be purely combinatorial (and thus, should not require access to a theorem prover to discharge the entailment), and so we use a *syntactic* formulation of entailment. Recalling that pre-condition and post-conditions of Hoare triples are sets of assertions interpreted conjunctively, $\varphi \supseteq \psi$ implies that φ entails ψ. The sequencing rule is formalized as follows:

$$\textsc{Sequencing}\quad \frac{\{\varphi_0\}\ \tau_0\ \{\varphi_1\} \qquad \varphi_1 \supseteq \varphi_1' \qquad \{\varphi_1'\}\ \tau_1\ \{\varphi_2\}}{\{\varphi_0\}\ \tau_0 ; \tau_1\ \{\varphi_2\}}$$

Suppose $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ is a program. The symmetry rule exploits the fact that thread identifiers in P are interchangeable, and therefore uniformly permuting thread identifiers in a valid Hoare triple yields another valid Hoare triple. For any permutation $\pi : N \to N$ of thread identifiers and any formula φ, we use $\varphi[\pi]$ to denote the result of substituting every local variable $\mathtt{x}(\mathtt{i})$ in φ with $\mathtt{x}(\pi(\mathtt{i}))$. The symmetry rule is formalized as follows:

$$\textsc{Symmetry}\quad \frac{\{\varphi\}\ \langle \sigma_1 : \mathtt{i}_1\rangle \cdots \langle \sigma_n : \mathtt{i}_n\rangle\ \{\psi\}}{\{\varphi[\pi]\}\ \langle \sigma_1 : \pi(\mathtt{i}_1)\rangle \cdots \langle \sigma_n : \pi(\mathtt{i}_n)\rangle\ \{\psi[\pi]\}} \quad \pi : N \to N \text{ is a permutation}$$

Lastly, the rule of conjunction allows one to combine two theorems about the same trace by conjoining preconditions and post-conditions:

$$\textsc{Conjunction}\quad \frac{\{\varphi_1\}\ \tau\ \{\psi_1\} \qquad \{\varphi_2\}\ \tau\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ \tau\ \{\psi_1 \wedge \psi_2\}}$$

Next, we formalize our notion of a proof space:

Definition 5.2.1 (Proof space)**.** *A* proof space $\mathscr{H}$ *is a set of valid Hoare triples that is closed under* Sequencing*,* Symmetry*, and* Conjunction*.*

The idea behind using proof spaces as correctness proofs can be summarized in the following proof rule. Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ be a program, and let ℓ_{err} be an error location. *If we can find a proof space* $\mathscr{H}$ *such that for every error trace* $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$*, the Hoare triple* $\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$ *belongs to* $\mathscr{H}$*, then* ℓ_{err} *is unreachable in* P*.*

Our main interest in proof spaces is in using the above proof rule as a foundation for software model checking. Towards this end, we augment proof spaces with additional conditions that make it possible to manipulate them algorithmically. We call the proof spaces satisfying these conditions *regular proof spaces*. First, we define *basic Hoare triples*, which are the "generators" of such proof spaces. The idea behind basic Hoare triples is that they should be as simple as possible. This is realized in three ways. First, for any basic Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$, τ always consists of exactly one command: if τ is any longer, then $\{\varphi\}\ \tau\ \{\psi\}$ can be generated from simpler Hoare triples via Sequencing. Second, for any basic Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$, ψ is a singleton: if ψ is a conjunction, then $\{\varphi\}\ \tau\ \{\psi\}$ can be generated from simpler Hoare triples via Conjunction. Third, for any basic Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$, the threads

that appear in the pre-condition φ should also appear in τ or the post-condition ψ (otherwise, they are irrelevant to the proof). Formally,

Definition 5.2.2 (Basic Hoare triple)**.** *A* basic Hoare triple *is a valid Hoare triple of the form* $\{\varphi\}\ \langle\sigma : \mathtt{i}\rangle\ \{\psi\}$ *where*

1. ψ *is a singleton*
2. *any thread that appears in the pre-condition* φ *is either* `i` *or appears in the post-condition ("the precondition mentions only relevant threads"). That is, for each thread* $\mathtt{j} \neq \mathtt{i} \in N$ *such that some local variable* $x(\mathtt{j})$ *of* `j` *appears in* φ*, some local variable* $y(\mathtt{j})$ *of* `j` *must appear in* ψ*.*

Definition 5.2.3 (Regular)**.** *We say that a proof space* $\mathscr{H}$ *is* regular *if there exists a finite set of basic Hoare triples* H *such that* $\mathscr{H}$ *is the smallest proof space that contains* H*. We call* H *a* basis *for* $\mathscr{H}$*.*

The remainder of this chapter studies some of the natural questions surrounding proof spaces. In the next section, we address the problem of how one may mechanically check proofs (*how can we be assured that a proof space proves a non-reachability property?*). In Section 5.4, we study the expressive power of proof spaces (*what can be proved using proof spaces?*). In Section 5.5, we show how proof spaces can be used in software model checking (*how can we generate proof spaces automatically?*). But before we address these questions, we will present an example of proof spaces in action.

5.2.1 Motivating Example

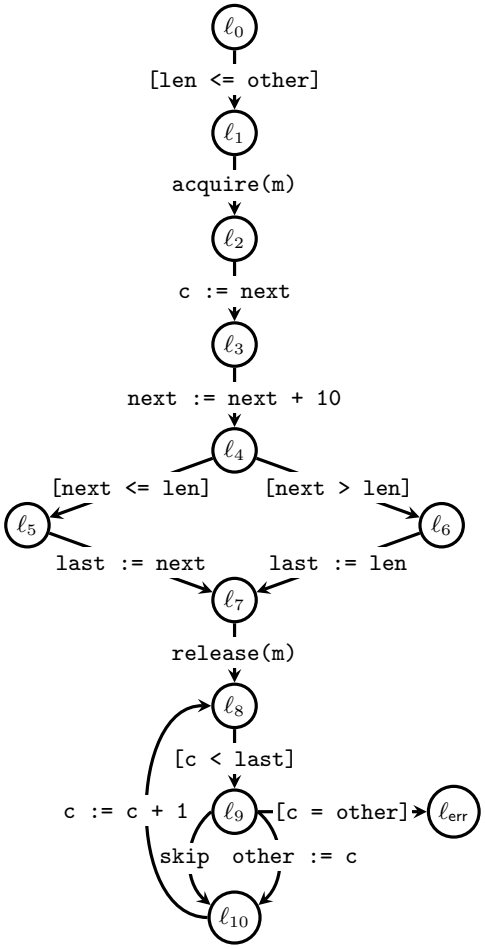
Consider a simplified implementation of a thread pool, where an arbitrary number of threads execute the code appearing in Figure 5.1a. The global variable `tasks` holds an array of tasks of length `len`, the global variable `next` stores the index of the next available task, and the global variable `m` is a lock that protects access to `next`. Each thread `i` has two local variables `c(i)` and `last(i)` that represent the current and last task in the block of tasks acquired by thread `i`. Each thread operates by acquiring a block of 10 consecutive tasks (lines 1-8) and then performing the tasks in its block in sequence (lines 9-13). The `else` branch (line 6) ensures that once the last of the `tasks` array is reached, all remaining threads that attempt to acquire a block of tasks acquire an empty block (`c = last`).

The property of interest in this program is that no two threads are assigned the same task. A PLIP program that encodes the essential features of this program is pictured in Figure 5.1b. The property of interest encoded as non-reachability of the error location ℓ_{err}. The encoding makes use of an additional global variable `other` that stores the position of a task that has been allocated to some ("other") thread. Initially, `other` is assumed to be greater than the length of the tasks array `len`, and whenever a thread works on a task it may non-deterministically choose to store that task into `other`. The error location is reached when a thread `i` finds that `other` contains the task `c(i)` that `i` is working on. Intuitively, the property holds because each thread `i` is assigned an interval of tasks $[\mathtt{c(i)}, \mathtt{last(i)})$, and the intervals assigned to different threads may not overlap (for all `j` not equal to `i`, we must have $\mathtt{last(i)} \leq \mathtt{c(j)}$ or $\mathtt{last(j)} \leq \mathtt{c(i)}$).

Proof space for thread pooling Figure 5.2 pictures a set of basic Hoare triples that generates a proof space for the thread pooling program. In the figure, the Hoare triples are categorized into six groups: *Disjoint*, *Invariance*, *Bound*, *Locking*, *Left*, and *Right*. The *Disjoint* group is responsible for

```
global int :  len                  // total number of tasks
global int array(len) :  tasks     // array of tasks
global int :  next   // position of next available task block
global lock :  m                   // lock protecting next
Thread:
 local int :  c;                   // position of current task
 local int :  last;     // position of last task in the block
// acquire block of tasks
1 acquire(m)
2   c := next
3   next := next + 10
4   if (next <= len):
5     last := next
6   else:
7     last := len
8   release(m)
// perform block of tasks
9   while (c < last):
//      work on task[c]
10    c := c + 1
```

(a) Pseudo-code for a simple thread pool.



(b) PLIP encoding

Figure 5.1: Thread pooling example, adapted from [Sanchez et al., 2012].

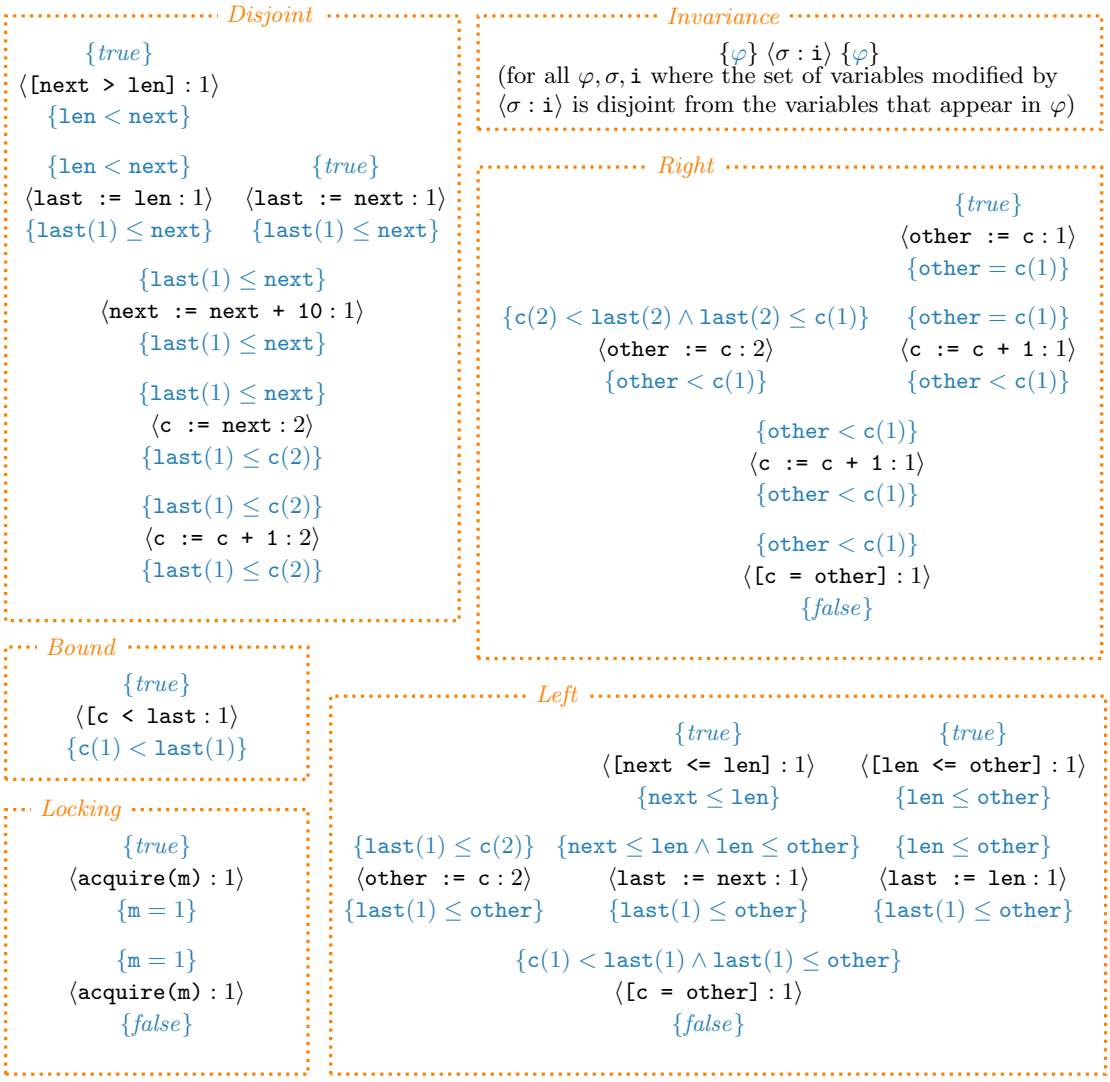


Figure 5.2: A basis for a proof space for the thread pooling example.

proving that threads are assigned disjoint intervals of tasks. The *Invariance* group is responsible for maintaining invariants under the action of irrelevant commands. The *Bound* triple is responsible for proving that a thread never exceeds the interval of tasks to which it is assigned. The *Locking* group is responsible for proving infeasibility of traces that violate locking semantics. Finally the *Left* and *Right* groups are responsible for proving infeasibility of error traces, either because the `other` task lies to the *left* or *right* of the task assigned to the thread that enters the error location.

We now demonstrate how the proof space is used to argue for the correctness of the program. We must show that for every trace τ of P that reaches ℓ_{err} *there exists* a derivation of $\{true\}\ \tau\ \{false\}$ that can be constructed from the triples in Figure 5.2 using the inference rules of SYMMETRY, CONJUNCTION, and SEQUENCING.

Let us first consider the pair of Hoare triples in the *Locking* group. Recall that we treat the `acquire(m)` command as the atomic sequence **atomic**{`[m = 0]; m := 1`}, and `release(m)` command as the assignment `m := 0`. The *Locking* Hoare triples encapsulate the reasoning required to prove that the lock `m` provides mutually exclusive access to the variable `next`. Any trace that violates the locking semantics can be proved infeasible using the Hoare triples in the *Locking* and *Invariance* groups along with the SEQUENCING and SYMMETRY rules. To see why, observe that any trace τ that violates locking semantics can be decomposed as follows:

$$\tau = \tau_1 \cdot \langle \texttt{acquire(m)} : \texttt{i} \rangle \cdot \tau_2 \cdot \langle \texttt{acquire(m)} : \texttt{j} \rangle \cdot \tau_3$$

where `m` is not released along τ_2. Observe that the following Hoare triples belong to the proof space generated by the basis in Figure 5.2:

$\{true\}$	τ_1	$\{true\}$	*Invariance*, SEQUENCING
$\{true\}$	$\langle \texttt{acquire(m)} : \texttt{i} \rangle$	$\{m = 1\}$	*Locking*
$\{m = 1\}$	τ_2	$\{m = 1\}$	*Invariance*, SEQUENCING
$\{m = 1\}$	$\langle \texttt{acquire(m)} : \texttt{j} \rangle$	$\{false\}$	*Locking*
$\{false\}$	τ_3	$\{false\}$	*Invariance*, SEQUENCING

By SEQUENCING the above triples, we may infer $\{true\}\ \tau\ \{false\}$. We conclude that for any trace τ that violates locking semantics, an infeasibility theorem for τ belongs to the proof space generated by the basis in Figure 5.2.

We are now free to concentrate on the error traces that *do* respect locking semantics. The Hoare triples in the *Disjoint* group can be used to show that if threads (say threads 2 and 9) both acquire a block of tasks along a trace τ, then either $\{true\}\ \tau\ \{\texttt{last}(2) \leq \texttt{c}(9)\}$ or $\{true\}\ \tau\ \{\texttt{last}(9) \leq \texttt{c}(2)\}$ may derived (depending on the order in which the threads acquired their tasks). An example of such a trace (which can be proved from the *Disjoint* and *Invariance* axioms using SEQUENCING and SYMMETRY) appears in Figure 5.3(a). We encourage the reader to convince themself that if the trace in Figure 5.3(a) is extended by the initialization sequence of a third thread (say, thread 5) to obtain a trace τ', then

$$\{true\}\ \tau'\ \{\texttt{last}(2) \leq \texttt{c}(9) \wedge \texttt{last}(2) \leq \texttt{c}(5) \wedge \texttt{last}(9) \leq \texttt{c}(5)\}$$

also belongs to the proof space (using the CONJUNCTION rule). More generally, one can see that the argument that threads are assigned disjoint intervals of tasks can be adapted to traces with any number of threads.

$\{true\}$
$\langle$[len <= other] : 2$\rangle$
$\langle$[len <= other] : 9$\rangle$
$\langle$acquire(m) : 2$\rangle$
$\langle$c := next : 2$\rangle$
$\langle$next := next + 10 : 2$\rangle$
$\langle$[next <= len] : 2$\rangle$
$\{true\}$
$\langle$last := next : 2$\rangle$
$\{\texttt{last}(2) \leq \texttt{next}\}$
$\langle$release(m) : 2$\rangle$
$\langle$acquire(m) : 9$\rangle$
$\{\texttt{last}(2) \leq \texttt{next}\}$
$\langle$c := next : 9$\rangle$
$\{\texttt{last}(2) \leq \texttt{c}(9)\}$
$\langle$next := next + 10 : 9$\rangle$
$\langle$[next <= len] : 9$\rangle$
$\langle$last := next : 9$\rangle$
$\langle$release(m) : 9$\rangle$
$\{\texttt{last}(2) \leq \texttt{c}(9)\}$

(a) Thread 2's tasks are disjoint from Thread 9's.

$\{\texttt{last}(2) \leq \texttt{c}(9)\}$
$\langle$[c < last] : 2$\rangle$
$\{\texttt{c}(2) < \texttt{last}(2) \wedge \texttt{last}(2) \leq \texttt{c}(9)\}$
$\langle$other := c : 2$\rangle$
$\{\texttt{other} < \texttt{c}(9)\}$
$\langle$[c < last] : 9$\rangle$
$\{\texttt{other} < \texttt{c}(9)\}$
$\langle$[c = other] : 9$\rangle$
$\{false\}$

(b) Infeasibility proof using *Left* axioms

$\{\texttt{last}(2) \leq \texttt{c}(9)\}$
$\langle$[c < last] : 9$\rangle$
$\{\texttt{last}(2) \leq \texttt{c}(9)\}$
$\langle$other := c : 9$\rangle$
$\{\texttt{last}(2) \leq \texttt{other}\}$
$\langle$[c < last] : 2$\rangle$
$\{\texttt{c}(2) < \texttt{last}(2) \wedge \texttt{last}(2) \leq \texttt{other}\}$
$\langle$[c = other] : 2$\rangle$
$\{false\}$

(c) Infeasibility proof using *Right* axioms

Figure 5.3: Example derivations of the proof space generated by Figure 5.2

Since threads are assigned disjoint intervals of tasks, the guard `[c = other]` leading to the error location is never satisfied: either `c` is strictly less than `other` or strictly greater. Figure 5.3(b) and (c) give two proofs to this effect (derivable from the *Left* and *Right* axioms respectively, along with the invariance axioms and the inference rules). By concatenating the trace in Figure 5.3(a) with the one in (b) (and similarly (c)), we arrive at an error trace of $\mathcal{L}(P, \ell_{\mathsf{err}})$ along with a derivation of its infeasibility.

The proof space proves infeasibility of not just these two example traces, but in fact *all* error traces of $\mathcal{L}(P, \ell_{\mathsf{err}})$. That is, for any error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$, it is possible to derive $\{true\}\ \tau\ \{false\}$ using the Symmetry, Sequencing, and Conjunction rules starting from the axioms given in Figure 5.2. The question is: how can one be assured that all (infinitely many) error traces can be proved infeasible in this way? In Section 5.3 we will show how to formalize such an argument, and moreover, give a procedure for checking that it holds.

5.3 Proof checking

This section tackles the question: *how we can convince ourselves that a proof space proves infeasibility of all traces of a program that violate a given non-reachability property?* Our solution to this problem begins by introducing a new class of automata, *predicate automata*, which can be used to represent both the set of error traces of a program and the set of traces τ such that $\{true\}\ \tau\ \{false\}$ belongs to a given regular proof space. We show that the problem of checking whether every error trace is proved infeasible by a given regular proof space can be reduced to the emptiness problem for predicate automata. Thus, predicate automata serve the same role for proof spaces that alternating finite automata serve for iDFGs (Section 4.3). We show that the emptiness problem is undecidable in general, but we give a

semi-algorithm and show that it is a decision procedure for an interesting class of predicate automata. Last, we give a mechanically verifiable system for proving that a predicate automaton accepts an empty language.

5.3.1 Predicate automata

Predicate automata are a class of infinite-state automata that recognize languages over an alphabet of the form $\Sigma \times N$, where Σ is a finite set of labels (program commands) and N is a possibly infinite set of indices (thread identifiers).[1] They can be understood informally by analogy with alternating finite automata. In the special case that the set of indices N is a singleton, there is in fact an exact correspondence between predicate automata and alternating finite automata, but generally we will be interested in the case that N is infinite. If one thinks of alternating finite automata as "propositional logic automata," one might think of predicate automata as "first-order logic automata." An alternating finite automaton is equipped with a finite set of states, and its transition function assigns to each state and each letter of its (finite) alphabet a positive Boolean formula with propositions drawn from the set of states. Conjuctions in such formulas are interpreted as universal choice and disjunction as existential choice. A predicate automaton (PA) is equipped with a relational first-order *vocabulary* $\langle Q, ar\rangle$ consisting of a finite set of predicate symbols Q and a function $ar : Q \to \mathbb{N}$ that maps each predicate symbol to its arity. A state of a PA is a proposition of the form $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})$, where $q \in Q$ is a predicate symbol and $\mathtt{i}_1, ..., \mathtt{i}_{ar(q)} \in N$ are indices. A PA's transition function assigns each predicate symbol q and each label σ a positive formula $\delta(q, \sigma)$ over the vocabulary $\langle Q, ar\rangle$. The transition function can be thought of as a symbolic representation of an infinite transition function that maps each proposition $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})$ and each letter $\langle \sigma : \mathtt{i}\rangle$ to the ground formula obtained by instantiating the free variables in the formula $\delta(q, \sigma)$ (the "formal parameters") with the indices $\mathtt{i}, \mathtt{i}_1, ..., \mathtt{i}_{ar(q)}$ (the "actual parameters").

We now define predicate automata. Fix a countable linearly ordered set of variable symbols $\{i_0, i_1, ...\}$, ranging over indices. Given a relational vocabulary $\langle Q, ar\rangle$, we define the set of *positive formulas* $\mathcal{F}(Q, ar)$ over $\langle Q, ar\rangle$ to be the set of quantifier-free negation-free formulas where each atom is either (1) a proposition of the form $q(i_{j_1}, ..., i_{j_n})$ (where $j_1, ..., j_n \in \mathbb{N}$), or (2) an equation $i_j = i_k$ (where $j, k \in \mathbb{N}$), or (3) a dis-equation $i_j \neq i_k$ (where $j, k \in \mathbb{N}$). The syntax of positive formulas is defined explicitly in Figure 5.4a. Predicate automata are defined as follows:

Definition 5.3.1 (Predicate automata)**.** *A* predicate automaton *(PA) is a 7-tuple*

$$A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$$

where

- $\langle Q, ar\rangle$ *is a relational vocabulary,*
- Σ *is a finite set of labels,*
- N *is a (possibly infinite) set of indices,*
- $\varphi_{\mathsf{start}} \in \mathcal{F}(Q, ar)$ *is an initial formula with no free variables,*
- $F \subseteq Q$ *is a set of accepting predicate symbols, and*

[1] Such languages are commonly called *data languages* [Neven et al., 2004].

$$\textsc{FProp}\ \frac{q \in Q}{q(i_{j_1}, ..., i_{j_{ar(q)}}) : \mathcal{F}(Q, ar)} \qquad \textsc{FEq}\ \frac{}{i_j = i_k : \mathcal{F}(Q, ar)} \qquad \textsc{FDisEq}\ \frac{}{i_j \neq i_k : \mathcal{F}(Q, ar)}$$

$$\textsc{FAnd}\ \frac{\varphi : \mathcal{F}(Q, ar) \qquad \psi : \mathcal{F}(Q, ar)}{\varphi \wedge \psi : \mathcal{F}(Q, ar)} \qquad \textsc{FOr}\ \frac{\varphi : \mathcal{F}(Q, ar) \qquad \psi : \mathcal{F}(Q, ar)}{\varphi \vee \psi : \mathcal{F}(Q, ar)}$$

(a) Positive formulas over the vocabulary $\langle Q, ar \rangle$ of a PA.

$$\textsc{GProp}\ \frac{q \in Q \qquad \mathtt{i}_1 \in N \qquad ... \qquad \mathtt{i}_{ar(q)} \in N}{q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) : \underline{\mathcal{F}}(Q, N, ar)} \qquad \textsc{GEq}\ \frac{\mathtt{i} \in N \qquad \mathtt{j} \in N}{\mathtt{i} = \mathtt{j} : \underline{\mathcal{F}}(Q, N, ar)} \qquad \textsc{GDisEq}\ \frac{\mathtt{i} \in N \qquad \mathtt{j} \in N}{\mathtt{i} \neq \mathtt{j} : \underline{\mathcal{F}}(Q, N, ar)}$$

$$\textsc{GAnd}\ \frac{\varphi : \mathcal{F}(Q, ar) \qquad \psi : \mathcal{F}(Q, ar)}{\varphi \wedge \psi : \underline{\mathcal{F}}(Q, N, ar)} \qquad \textsc{GOr}\ \frac{\varphi : \mathcal{F}(Q, ar) \qquad \psi : \mathcal{F}(Q, ar)}{\varphi \vee \psi : \underline{\mathcal{F}}(Q, N, ar)}$$

(b) Ground positive formulas over the ground vocabulary $\langle Q, N, ar \rangle$ of a PA.

Figure 5.4: Syntax for positive and ground formulas over the vocabulary of a predicate automaton

- $\delta : Q \times \Sigma \to \mathcal{F}(Q, ar)$ *is a transition function that maps each predicate symbol* $q \in Q$ *and each label* $\sigma \in \Sigma$ *to a positive formula* $\delta(q, \sigma)$ *such that the free variables of* $\delta(q, \sigma)$ *are members of the set* $\{i_0, ..., i_{ar(q)}\}$*. One may think of* δ *as a collection of symbolic rewrite rules of the form*

$$q(i_1, ..., i_{ar(q)}) \xrightarrow{\sigma : i_0} \delta(q, \sigma) \ ,$$

so the free variable restriction corresponds to the constraint that the free variables on the right hand side of the rule are bound on the left hand side.

For better readability, we will typically write a transition function δ of a PA in a form that makes the (implicit) formal parameters explicit: for example, instead of

$$\delta(q, \sigma) = (i_0 \neq i_1 \wedge (q(i_0, i_1) \vee q(i_1, i_2))) \vee (i_0 = i_1 \wedge q(i_1, i_2) \wedge q(i_2, i_1))$$

we will typically write

$$\delta(q(i, j), \langle \sigma : k \rangle) = (k \neq i \wedge (q(k, i) \vee q(i, j))) \vee (k = i \wedge q(i, j) \wedge q(j, i)) \ .$$

We define the language recognized by a predicate automaton similarly to the way that we defined the language recognized by an alternating finite automaton (Definition 4.3.3).

Definition 5.3.2 (PA semantics)**.** *Let* $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F \rangle$ *be a predicate automaton. Define* $\underline{\mathcal{F}}(Q, N, ar)$ *to be the set of* ground formulas *(without free variables) in the vocabulary* $\langle Q, N, ar \rangle$ *consisting of the relational vocabulary* $\langle Q, ar \rangle$ *plus the constant symbols* N *(the syntax of ground formulas is given explicitly in Figure 5.4b). Define a function* $\mathcal{L}_A : \underline{\mathcal{F}}(Q, N, ar) \to 2^{\Sigma(N)^*}$ *mapping ground formulas*

to trace languages to be the least function (in point-wise subset inclusion order) such that:

$$\begin{aligned}
\epsilon \in \mathcal{L}_A(q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})) &\iff q \in F \\
\tau\langle\sigma : \mathtt{i}\rangle \in \mathcal{L}_A(q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})) &\iff \tau \in \mathcal{L}_A(\delta(q, \sigma)[i_0 \mapsto \mathtt{i}, i_1 \mapsto \mathtt{i}_1, ..., i_n \mapsto \mathtt{i}_n]) \\
\tau \in \mathcal{L}_A(\mathtt{i} = \mathtt{j}) &\iff \mathtt{i} = \mathtt{j} \\
\tau \in \mathcal{L}_A(\mathtt{i} \neq \mathtt{j}) &\iff \mathtt{i} \neq \mathtt{j} \\
\tau \in \mathcal{L}_A(\varphi \wedge \psi) &\iff \tau \in \mathcal{L}_A(\varphi) \text{ and } \tau \in \mathcal{L}(\psi) \\
\tau \in \mathcal{L}_A(\varphi \vee \psi) &\iff \tau \in \mathcal{L}_A(\varphi) \text{ or } \tau \in \mathcal{L}_A(\psi)
\end{aligned}$$

Define $\mathcal{L}(A)$, the language recognized by A, to be $\mathcal{L}_A(\varphi_{\mathsf{init}})$.

It will often be useful, particularly when we describe our semi-algorithm for checking emptiness of predicate automata, to use a more operational definition of the language recognized by predicate automaton. We may associate with any predicate automaton a nondeterministic transition system with transitions labelled by letters of its alphabet. The states of the transition system are the *configurations* of the automaton:

Definition 5.3.3 (Configuration)**.** *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. A* configuration $\mathcal{C}$ *of A is finite set of ground propositions of the form $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})$, where $q \in Q$ and $\mathtt{i}_1, ..., \mathtt{i}_{ar(q)} \in N$. Equivalently, the configurations of A are the finite structures over the relational vocabulary $\langle Q, ar\rangle$, in the usual sense of first-order logic.*

We now define the transitions of the automaton. Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. For any ground formula $\varphi \in \underline{\mathcal{F}}(Q, N, ar)$, we let $\mathsf{Cubes}(\varphi)$ be the set of cubes in the disjunctive normal form of φ. We may think of $\mathsf{Cubes}(\varphi)$ as the set of "minimal" configurations that satisfy φ. $\mathsf{Cubes}(\varphi)$ is defined explicitly as follows:

$$\begin{aligned}
\mathsf{Cubes}(q(\mathtt{i}_1, ..., \mathtt{i}_n)) &\triangleq \{\{q(\mathtt{i}_1, ..., \mathtt{i}_n)\}\} \\
\mathsf{Cubes}(\mathtt{i} = \mathtt{j}) &\triangleq \begin{cases} \{\emptyset\} & \text{if } \mathtt{i} = \mathtt{j} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{Cubes}(\mathtt{i} \neq \mathtt{j}) &\triangleq \begin{cases} \{\emptyset\} & \text{if } \mathtt{i} \neq \mathtt{j} \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{Cubes}(\varphi \wedge \psi) &= \{\mathcal{C} \cup \mathcal{C}' : \mathcal{C} \in \mathsf{Cubes}(\varphi) \wedge \mathcal{C}' \in \mathsf{Cubes}(\psi)\} \\
\mathsf{Cubes}(\varphi \vee \psi) &\triangleq \mathsf{Cubes}(\varphi) \cup \mathsf{Cubes}(\psi)
\end{aligned}$$

For any configuration $\mathcal{C}$ and any letter $\langle\sigma : \mathtt{i}\rangle$, we define the set of $\mathcal{C}$'s successors as:

$$\mathsf{Post}_A(\mathcal{C}, \sigma : \mathtt{i}) \triangleq \mathsf{Cubes}\Big(\bigwedge\{\delta(q, \sigma)[i_0 \mapsto \mathtt{i}, i_1 \mapsto \mathtt{i}_1, ..., i_{ar(q)} \mapsto \mathtt{i}_{ar(q)}] : q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{C}\}\Big) \ .$$

We say that a configuration $\mathcal{C}$ transitions to another $\mathcal{C}'$ upon reading $\langle\sigma : \mathtt{i}\rangle$, written $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, if and only if $\mathcal{C}' \in \mathsf{Post}_A(\mathcal{C}, \sigma : \mathtt{i})$.

A configuration $\mathcal{C}$ is *initial* if and only if $\mathcal{C} \in \mathsf{Cubes}(\varphi_{\mathsf{init}})$. A configuration is *accepting* if for all propositions $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{C}$, we have that $q \in F$ is an accepting predicate symbol; otherwise, it is

rejecting.

The connection between the language recognized by a predicate automaton (Definition 5.3.2) and its associated transition system is as follows.

Proposition 5.3.4. *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. A trace $\tau = \langle \sigma_1 : \mathtt{i}_1\rangle \cdots \langle \sigma_n : \mathtt{i}_n\rangle \in \Sigma(N)^*$ belongs to $\mathcal{L}(A)$ if and only if there exists is a sequence of configurations $\mathcal{C}_n, ..., \mathcal{C}_0$ (an* accepting sequence*) such that $\mathcal{C}_n$ is initial, $\mathcal{C}_0$ is accepting, and for each $r \in \{1, ..., n\}$, $\mathcal{C}_r \xrightarrow{\sigma_r : \mathtt{i}_r} \mathcal{C}_{r-1}$.*

It is important to note that the definition of accepting sequences above implies that predicate automata read their input from right to left rather than left to right. This simplifies our presentation by obviating the need for language reversals that would otherwise be necessary (cf. the case of recognizing iDFG languages with alternating finite automata, Section 4.3).

5.3.2 Recognizing error traces

Our first example of a predicate automaton will be the one constructed to accept the language of traces of a program that violate some given non-reachability property. The construction can be seen as a generalization of the alternating finite automaton construction in Section 4.3.2 to an arbitrary set of threads. Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ be a program and let ℓ_{err} be an error location. We construct the predicate automaton $\mathcal{A}(P, \ell_{\mathsf{err}}) = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ that recognizes $\mathcal{L}(P, \ell_{\mathsf{err}})$ as follows:

- $Q = \{I, \mathsf{err}\} \cup \mathsf{Loc}$, where I and err are distinguished predicate symbols not belonging to Loc. I and err are nullary predicate symbols, and each $\ell \in \mathsf{Loc}$ is monadic. We construct $\mathcal{A}(P, \ell_{\mathsf{err}})$ so that
  - a trace τ is accepted starting from the state $\ell(\mathtt{i})$ (i.e., $\tau \in \mathcal{L}_{\mathcal{A}(P,\ell_{\mathsf{err}})}(\ell(\mathtt{i}))$) iff $\tau|_{\mathtt{i}}$ corresponds to a control flow path of P from ℓ_{init} to ℓ, where $\tau|_{\mathtt{i}}$ is the trace obtained from τ by deleting every command *not* executed by $\mathtt{i}$.
  - a trace τ is accepted starting from I (which we call the *initializer* state) iff τ is an interleaved control flow path of P (from the initial control state loc_{init} to any control state).
  - a trace τ is accepted starting from err iff there exists a thread $\mathtt{i}$ such that $\tau|_{\mathtt{i}}$ is a control flow path of P from ℓ_{init} to ℓ_{err}.
- The transition function is defined as follows:

$$\delta(\ell(i), \langle \sigma : j\rangle) \triangleq \begin{cases} \overbrace{(i = j \wedge \mathsf{src}(\sigma)(i))}^{\textit{Follow } \sigma \textit{ backwards}} \vee \overbrace{(i \neq j \wedge \ell(i))}^{\textit{Ignore other threads}} & \text{if } \ell = \mathsf{tgt}(\sigma) \\ \underbrace{i \neq j \wedge \ell(i)}_{\textit{Ignore other threads}} & \text{if } \ell \neq \mathsf{tgt}(\sigma) \end{cases}$$

$$\delta(I, \langle \sigma : i\rangle) \triangleq I \wedge \mathsf{src}(\sigma)(i) \qquad \textit{Remain on } I \textit{ \& initialize thread } i\textit{'s program counter}$$

$$\delta(\mathsf{err}, \langle \sigma : i\rangle) \triangleq \begin{cases} \mathsf{err} \vee \mathsf{src}(\sigma)(i) & \text{if } \mathsf{tgt}(\sigma) = \ell_{\mathsf{err}} \quad \textit{Follow edge backwards if it leads to } \ell_{err} \\ \mathsf{err} & \text{if } \mathsf{tgt}(\sigma) \neq \ell_{\mathsf{err}} \quad \textit{Ignore transitions that do not} \end{cases}$$

- $\varphi_{\mathsf{init}} \triangleq I \wedge \mathsf{err}$ *(All threads have valid control flow, and some thread is at the error location)*
- $F \triangleq \{\ell_{\mathsf{init}}, I\}$ *(Accept when every thread is at ℓ_{init}; the initializer I is persistent)*

This construction yields the following proposition.

Proposition 5.3.5. *Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ be a program and let $\ell_{\mathsf{err}} \in \mathsf{Loc}$ be a control location. $\mathcal{A}(P, \ell_{\mathsf{err}})$ recognizes language of error traces $\mathcal{L}(P, \ell_{\mathsf{err}})$.*

5.3.3 Recognizing proof space languages

Let $\mathscr{H}$ be a regular proof space with basis H. We now show how to construct a predicate automaton $\mathcal{A}(H)$ that recognizes the set of traces τ such that $\{true\}\ \tau\ \{false\} \in \mathscr{H}$. The predicate automaton $\mathcal{A}(H)$ closely mirrors the structure of H: the predicates of $\mathcal{A}(H)$ correspond to the assertions in H, and each Hoare triple in H corresponds to a transition.

First we give some auxiliary definitions that will allow us to define $\mathcal{A}(H)$ succinctly. The SYMMETRY rule of proof spaces suggests that we should not distinguish between assertions that are identical up to renaming thread identifiers. For any formula φ, let $\mathsf{tid}(\varphi)$ be the sequence of the thread identifiers that appear in φ, in the order of their first occurrence from left to right. For example, we have $\mathsf{tid}(\mathtt{m}(3) < \mathtt{m}(2)) = 3, 2$. Let φ be a formula and $\mathtt{i}_1, ..., \mathtt{i}_n = \mathsf{tid}(\varphi)$. We define the *canonical name* $[\varphi]$ of φ to be the formula obtained by replacing each thread identifier $\mathtt{i}_j$ with j. For example, we have $[\mathtt{m}(3) < \mathtt{m}(2)] = \mathtt{m}(1) < \mathtt{m}(2)$. We will construct the predicate automaton $\mathcal{A}(H)$ so that its vocabulary consists of the set of all canonical names of assertions that appear H, where the arity of a canonical name $[\varphi]$ is the number of distinct thread identifiers that appear in φ.

Next, we show how a basic Hoare triple corresponds to a transition of a predicate automaton. For a concrete example, consider the Hoare triple

$$\{\mathtt{t(3)} \geq 0 \wedge \mathtt{t(9)} > \mathtt{t(3)}\}\ \langle \mathtt{t\ :=\ 2*t} : 9\rangle\ \{\mathtt{t(9)} > \mathtt{t(3)}\}\ .$$

This triple corresponds to the transition

$$\delta([\mathtt{t(1)} > \mathtt{t(2)}](i, j), \sigma : k) = [\mathtt{t(1)} \geq 0](j) \wedge [\mathtt{t(1)} > \mathtt{t(2)}](k, j) \wedge i \neq j \wedge i = k \wedge j \neq k\ .$$

Notice that the substitution instances of this transition (instantiating the variables i, j, and k with concrete indices) correspond exactly to the set of Hoare triples that can be derived via the SYMMETRY rule. In particular, note that the equations and dis-equations between thread variables enforce that there are no valid substitution instances that do *not* correspond to permutations of thread identifiers.

Let $\{\psi_1 \wedge \cdots \wedge \psi_n\}\ \langle \sigma : \mathtt{i}\rangle\ \{\varphi\} \in H$ be a basic Hoare triple. Translating this Hoare triple into a PA transition requires that we *generalize* it by replacing the concrete thread identifiers by symbolic thread variables. Let us use $\mathtt{i}_0 = \mathtt{i}$, to denote the identifier of the thread executing σ, and let $\mathtt{i}_1, ..., \mathtt{i}_n = \mathsf{tid}(\varphi)$ be a list of the thread identifiers that appear in the post condition $\mathtt{i}$. We will replace each concrete thread identifier $\mathtt{i}_s$ (which belongs to the set of threads N) with the thread variable i_s (which belongs to a syntactic sort of thread *variables* which range over N, but do not belong to it – note the difference in the type face). For any pre-condition assertion ψ_j in the Hoare triple we use $\vec{i_j}$ to denote a list of thread variables obtained by letting $\mathtt{k}_1, ..., \mathtt{k}_m = \mathsf{tid}(\psi_j)$ be the list of concrete thread identifiers appearing and ψ_j and taking $\vec{i_j} \triangleq i_{q_1}, ..., i_{q_m}$, where q_1 is the least number in $\{0, ..., n\}$ such that $\mathtt{k}_1 = \mathtt{i}_{q_1}$.[2] For any

[2]By condition 2 in the definition of Basic Hoare triples (Definition 5.2.2), the thread identifiers that appear in ψ_j are a subset of the ones that appear in φ and $\mathtt{i}$, so each of $\mathtt{k}_1, ..., \mathtt{k}_m$ belongs to the list $\mathtt{i}_0, ..., \mathtt{i}_n$.

pair of thread variables i_s and i_t, define

$$\mathsf{eq}(i_s, i_t) \triangleq \begin{cases} i_s = i_t & \text{if } \mathtt{i}_s = \mathtt{i}_t \\ i_s \neq i_t & \text{otherwise.} \end{cases} \qquad \textit{Note the difference in type face}$$

Define a function generalize that produces the right-hand-side of the PA transition rule corresponding to a Hoare triple as follows:

$$\mathsf{generalize}(\varphi, \mathtt{i}, \psi_1 \wedge \cdots \wedge \psi_n) \triangleq [\psi_1](\vec{i}_1) \wedge \cdots \wedge [\psi_n](\vec{i}_n) \wedge \bigwedge \{\mathsf{eq}(i_s, i_t) : 0 \leq s < t \leq n\} \ .$$

Finally, we are ready to define the predicate automaton $\mathcal{A}(H) = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F \rangle$ that recognizes the traces that can be proved infeasible from the basis H:

- $Q \triangleq \{[\varphi] : \exists \{\psi_1 \wedge \cdots \wedge \psi_n\}\ \langle \sigma : \mathtt{i} \rangle\ \{\psi\} \in H \wedge \varphi \in \{\psi_1, ..., \psi_n, \psi\}\}$ is the set of canonical names of assertions that appear in H. We construct $\mathcal{A}(H)$ so that for each state $[\varphi](\mathtt{i}_1, ..., \mathtt{i}_n)$, a trace τ is accepted starting at $[\varphi](\mathtt{i}_1, ..., \mathtt{i}_n)$ if and only if $\{true\}\ \tau\ \{\varphi[1 \mapsto \mathtt{i}_1, ..., n \mapsto \mathtt{i}_n]\}$ belongs to $\mathscr{H}$.
- $ar([\varphi]) \triangleq |\mathsf{tid}(\varphi)|$ assigns every canonical name $[\varphi]$ an arity equal to the number of distinct thread identifiers that appear in φ.
- $\delta([\varphi], \sigma) \triangleq \bigvee \{\mathsf{generalize}(\varphi, \mathtt{i}, \psi_1 \wedge \cdots \wedge \psi_n) : \exists \psi.[\psi] = [\varphi]$ and $\{\psi_1 \wedge \cdots \wedge \psi_n\}\ \langle \sigma : \mathtt{i} \rangle\ \{\psi\} \in H\}$. Each Hoare triple in H corresponds to a transition from its post-condition to its pre-condition, and if there are several Hoare triples in H with the same (canonically named) post-condition, then we combine them via disjunction.
- $\varphi_{\mathsf{init}} \triangleq [false]$ ($\mathcal{A}(H)$ accepts τ when $\{true\}\ \tau\ \{false\}$ belongs to $\mathscr{H}$).
- $F \triangleq \emptyset$. (There are no accepting predicates ($\mathcal{A}(H)$ accepts by transitioning to *true*)).

The construction $\mathcal{A}(H)$ gives way to the following result:

Proposition 5.3.6. *Let $\mathscr{H}$ be a regular proof with basis H. $\mathcal{A}(H)$ recognizes exactly the set of traces τ such that $\{true\}\ \tau\ \{false\} \in \mathscr{H}$.*

5.3.4 Mechanical verification of proof spaces

Propositions 5.3.5 and 5.3.6 together imply that the problem of checking whether a proof space proves that every trace of a program that violates a given non-reachability property is infeasible can be reduced to the language inclusion problem for predicate automata. The following proposition reduces the problem further to the emptiness problem (noting that $\mathcal{L}(\mathcal{A}(P, \ell_{\mathsf{err}})) \subseteq \mathcal{L}(\mathcal{A}(H))$ if and only if the intersection of $\mathcal{L}(\mathcal{A}(P, \ell_{\mathsf{err}}))$ with the complement of $\mathcal{L}(\mathcal{A}(H))$ is empty):

Proposition 5.3.7. *Predicate automata languages are closed under intersection and complement.*

Proof. The constructions for intersection and complementation of predicate automata follow the classical ones for alternating finite automata.

Let A and A' be PAs. We form their intersection $A \cap A'$ by taking the vocabulary to be the disjoint union of the vocabularies of A and A', and define the transition relation and accepting predicates accordingly. The initial formula is obtained by conjoining the initial formulas of A and A'.

Given a PA $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$, we form its complement $\overline{A} = \langle \overline{Q}, \overline{ar}, \Sigma, N, \overline{\delta}, \overline{\varphi_{\mathsf{init}}}, \overline{F}\rangle$ as follows. We define the vocabulary $(\overline{Q}, \overline{ar})$ to be a "negated copy" of (Q, ar): $\overline{Q} \triangleq \{\overline{q} : q \in Q\}$ and $\overline{ar}(\overline{q}) \triangleq ar(q)$. The set of accepting predicate symbols is the (negated) set of rejecting predicate symbols from A: $\overline{F} \triangleq \{\overline{q} \in \overline{Q} : q \in Q \setminus F\}$. For any formula φ in $\mathcal{F}(Q, ar)$ in the vocabulary of A, we use $\overline{\varphi}$ to denote the "De Morganization" of φ in the vocabulary of $\overline{A}$, defined recursively by:

$$\overline{q(i_{j_1}, ..., i_{j_{ar(q)}})} \triangleq \overline{q}(i_{j_1}, ..., i_{j_{ar(q)}}) \qquad \overline{i_j = i_k} \triangleq i \neq j \qquad \overline{\varphi \wedge \psi} \triangleq \overline{\varphi} \vee \overline{\psi}$$
$$\overline{i_j \neq i_k} \triangleq i = j \qquad \overline{\varphi \vee \psi} \triangleq \overline{\varphi} \wedge \overline{\psi}$$

We define the transition function and initial formula of $\overline{A}$ by De Morganization: $\overline{\delta}(\overline{q}, \sigma)$ is defined to be $\overline{\delta(q, \sigma)}$ and the initial formula is defined to be $\overline{\varphi_{\mathsf{init}}}$. □

Checking emptiness for predicate automata

We give a semi-algorithm for checking PA emptiness that is sound (when the procedure terminates, it gives the correct answer) and complete for counter-examples (if the PA accepts a word, the procedure terminates). Our procedure for checking PA language emptiness is inspired by coverability algorithms for well-structured transition systems [Finkel, 1987, Abdulla et al., 1996, Finkel and Schnoebelen, 2001]. The algorithm is essentially a state-space exploration of a predicate automaton (starting from an initial configuration, searching for a reachable accepting configuration), but with one crucial improvement: we define a *covering* pre-order on configurations, and prune the search space by exploring only those configurations that are minimal with respect to this order. We show that in some interesting cases, namely for *monadic* predicate automata, this pruning strategy is sufficient to ensure termination of the search, despite the fact that the search space is infinite.

We begin by defining the covering relation on PA configurations:

Definition 5.3.8 (Covering)**.** *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a predicate automaton. Define the* covering pre-order $\preceq$ *on the configurations of A as follows: if $\mathcal{C}$ and $\mathcal{D}$ are configurations of A, then $\mathcal{C} \preceq \mathcal{D}$ ("$\mathcal{C}$* covers *$\mathcal{D}$") if there is a permutation $\pi : N \to N$ such that for all ground propositions $q(\mathtt{i}_1, .., \mathtt{i}_{ar(q)})$ belonging to $\mathcal{C}$, the renamed proposition $q(\pi(\mathtt{i}_1), ..., \pi(\mathtt{i}_{ar(q)}))$ belongs to $\mathcal{D}$. We call such a permutation π an* embedding *of $\mathcal{C}$ into $\mathcal{D}$.*

The idea behind the pruning strategy is that if $\mathcal{C}$ and $\mathcal{D}$ are configurations such that $\mathcal{C}$ covers $\mathcal{D}$, then any accepting path starting from $\mathcal{D}$ can be transformed into an accepting path starting from $\mathcal{C}$. Thus, if $\mathcal{C}$ is in the search space of the algorithm it is safe to remove $\mathcal{D}$ from the search. The argument follows from the following lemma:

Lemma 5.3.9 (Downwards compatibility)**.** *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA and let $\mathcal{C}$ and $\mathcal{D}$ be configurations of A such that $\mathcal{C} \preceq \mathcal{D}$. Then we have the following:*

1. *If $\mathcal{D}$ is accepting, then $\mathcal{C}$ is accepting.*

2. *For any $\langle \sigma : \mathtt{j}\rangle \in \Sigma \times N$ and $\mathcal{D}'$ such that $\mathcal{D} \xrightarrow{\sigma:\mathtt{j}} \mathcal{D}'$, there exists a configuration $\mathcal{C}$ and an index $\mathtt{i} \in N$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$ and $\mathcal{C}' \preceq \mathcal{D}'$.*

Proof. Let $\mathcal{A} = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA, let $\mathcal{C}$ and $\mathcal{D}$ be configurations of $\mathcal{A}$, and let $\pi : N \to N$ be an embedding of $\mathcal{C}$ into $\mathcal{D}$. Write $\mathcal{C}$ as

$$\mathcal{C} = \{q_1(\mathtt{i}_{1,1}, ..., \mathtt{i}_{1,ar(q_1)}), ..., q_n(\mathtt{i}_{n,1}, ..., \mathtt{i}_{n,ar(q_n)})\} .$$

By virtue of π being an embedding, we can write $\mathcal{D}$ as

$$\begin{aligned}\mathcal{D} = &\{q_1(\mathtt{j}_{1,1}, ..., \mathtt{j}_{1,ar(q_1)}), ..., q_n(\mathtt{j}_{n,1}, ..., \mathtt{j}_{n,ar(q_n)}\}\\ &\cup \{q_{n+1}(\mathtt{j}_{n+1,1}, ..., \mathtt{j}_{n+1,ar(q_1)}), ..., q_m(\mathtt{j}_{m,1}, ..., \mathtt{j}_{m,ar(q_m)})\}\end{aligned}$$

where $\mathtt{j}_{k,l} = \pi(\mathtt{i}_{k,l})$ for all $k \in \{1, ..., n\}$ and $l \in \{1, ..., ar(q_k)\}$.

1. Suppose that $\mathcal{D}$ is accepting. Then $q_1, ..., q_m$ must be accepting, so $\mathcal{C}$ is accepting.

2. Let $\langle \sigma : \mathtt{j}\rangle \in \Sigma(N)$, and let $\mathcal{D}'$ be a configuration such that $\mathcal{D} \xrightarrow{\sigma:\mathtt{j}} \mathcal{D}'$. Since $\mathcal{D}$ transitions to $\mathcal{D}'$ on reading $\langle \sigma : \mathtt{j}\rangle$, we have

   $$\mathcal{D}' \in \mathsf{Post}_A(\mathcal{D}, \sigma : \mathtt{j}) = \mathsf{Cubes}\Big(\bigwedge\{\delta(q, \sigma)[i_0 \mapsto \mathtt{j}, i_1 \mapsto \mathtt{i}_1, ..., i_{ar(q)} \mapsto \mathtt{i}_{ar(q)}] : q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{D}\}\Big) .$$

   Thus for each $k \in \{1, ..., m\}$, there is some

   $$\mathcal{D}'_k \in \mathsf{Cubes}(\delta(q_k, \sigma)[i_0 \mapsto \mathtt{i}, i_1 \mapsto \mathtt{i}_{k,1}, ..., i_{ar(q)} \mapsto \mathtt{i}_{ar(k,q_k)}])$$

   such that $\mathcal{D}'_k \subseteq \mathcal{D}'$.

   For each $k \in \{1, ..., n\}$, define $\mathcal{C}'_k \triangleq \{q(\pi^{-1}(\mathtt{i}_1), ..., \pi^{-1}(\mathtt{i}_{ar(q)})) : q(\mathtt{i}_1, ..., \mathtt{i}_n) \in \mathcal{D}'_k\}$, and note that we have

   $$\mathcal{C}'_k \in \mathsf{Cubes}(\delta(q_k, \sigma)[i_0 \mapsto \pi^{-1}(\mathtt{j}), i_1 \mapsto \pi^{-1}(\mathtt{i}_{k,1}), ..., i_{ar(q)} \mapsto \pi^{-1}(\mathtt{i}_{k,ar(q_k)})])$$

   Define $\mathcal{C}' \triangleq \mathcal{C}'_1 \cup ... \cup \mathcal{C}'_n$. Then we have $\mathcal{C} \xrightarrow{\sigma:\pi^{-1}(\mathtt{j})} \mathcal{C}'$ and $\mathcal{C}' \preceq \mathcal{D}'$ (π is an embedding). □

We now develop our algorithm in more detail. In the remainder of this section, let us fix a predicate automaton $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$. Without loss of generality, identify the set of threads N with the natural numbers. State-space exploration of A is complicated by the fact that the transition system associated with A is infinitely branching. However, for each fixed letter $\langle \sigma : \mathtt{i}\rangle$, a configuration has only finitely many successors, so the problem is only that the size of the alphabet is infinite. The key to solving this problem is to observe that all but finitely many indices $\mathtt{i} \in N$ are indistinguishable from the perspective of a given configuration. With this in mind, let us define the *support* $\mathsf{support}(\mathcal{C})$ of a configuration $\mathcal{C}$ to be the set of all indices that appear in $\mathcal{C}$; formally,

$$\mathsf{support}(\mathcal{C}) \triangleq \{\mathtt{i}_r : q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{C}, 1 \le r \le ar(q)\} .$$

If $\mathtt{i}$ and $\mathtt{j}$ are distinct indices *not* belonging to the support of $\mathcal{C}$, then $\mathtt{i}$ and $\mathtt{j}$ are effectively indistinguishable starting from $\mathcal{C}$. This intuition is formalized in the following lemma:

Lemma 5.3.10. *Let $\mathcal{C}$ be a configuration of A, $\mathtt{k}_1, \mathtt{k}_2 \in N \setminus \mathsf{support}(\mathcal{C})$, and $\sigma \in \Sigma$. For all configurations $\mathcal{C}_1$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{k}_1} \mathcal{C}_1$, there exists a configuration $\mathcal{C}_2$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{k}_2} \mathcal{C}_2$ and $\mathcal{C}_1 \preceq \mathcal{C}_2$ and $\mathcal{C}_2 \preceq \mathcal{C}_1$.*

As a result of this lemma, from a given configuration $\mathcal{C}$, it is sufficient to explore $\langle \sigma : \mathtt{i} \rangle$ such that the index $\mathtt{i}$ belongs to the support of $\mathcal{C}$, plus one additional index $\mathtt{j}$ not in $\mathcal{C}$'s support. Since our algorithm assumes that the set of thread identifiers is identified with the set of naturals, we may simply choose the additional index $\mathtt{j}$ to be 1 more than the maximum index in $\mathsf{support}(\mathcal{C})$.

Finally, we state a semi-algorithm for PA emptiness in Algorithm 5. The procedure operates by expanding a reachability forest $\langle V, E \rangle$ where the vertices (V) are configurations and the edges (E) are labelled by indexed letters. The frontier of the reachability tree is kept in a worklist *worklist*, and the set of *closed* nodes (configurations that have already been expanded) is kept in *Closed*.

Input : Predicate automaton $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F \rangle$
Output: *Empty*, if $\mathcal{L}(A)$ is empty; a word $w \in \mathcal{L}(A)$, if not
$Closed \leftarrow \emptyset$;
$V \leftarrow \emptyset$;
$E \leftarrow \emptyset$;
$worklist \leftarrow \mathsf{Cubes}(\varphi_{\mathsf{init}})$;
while $worklist \neq []$ **do**
  $\mathcal{C} \leftarrow head(worklist)$;
  $worklist \leftarrow tail(worklist)$;
  if $\neg \exists \mathcal{D} \in Closed$ *s.t.* $\mathcal{D} \preceq \mathcal{C}$ **then**
    /* Expand $\mathcal{C}$ */
    foreach $\mathtt{i} \in support(\mathcal{C}) \cup \{1 + \max support(\mathcal{C})\}$ **do**
      foreach $\sigma \in \Sigma$ **do**
        foreach $\mathcal{C}' \in Post_A(\mathcal{C}, \langle \sigma : \mathtt{i} \rangle)$ *and* $\mathcal{C}' \notin V$ **do**
          $V \leftarrow V \cup \{\mathcal{C}'\}$;
          $E \leftarrow E \cup \{\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'\}$;
          if $\mathcal{C}'$ *is accepting* **then**
            return *word w labelling the path in the graph (V, E) from $\mathcal{C}'$ to a root*;
          else
            Add $\mathcal{C}'$ to the end of *worklist*
          end
        end
      end
    end
  end
  $Closed \leftarrow Closed \cup \{\mathcal{C}\}$;
end
return *Empty*

Algorithm 5: Emptiness check for predicate automata

Theorem 5.3.11 (Partial correctness)**.** *Algorithm 5 is sound in the sense that if Algorithm 5 returns Empty, then $\mathcal{L}(A)$ is $\emptyset$, and if Algorithm 5 returns a trace τ, then $\tau \in \mathcal{L}(A)$.*

Theorem 5.3.12. *Algorithm 5 is complete for non-emptiness: if $\mathcal{L}(A)$ is nonempty, then Algorithm 5 returns a word in $\mathcal{L}(A)$.*

5.3.5 Decidability results

Although Algorithm 5 is sound and complete for non-emptiness, it is *not* complete for emptiness: Algorithm 5 may fail to terminate in the case that the language of the input PA is empty. In fact, this must be the case for any algorithm, because PA emptiness is undecidable in the general case:

Proposition 5.3.13. *General PA emptiness is undecidable.*

Proof. We reduce the halting problem for Minsky machines to PA emptiness.

Recall the definition of Minsky machines:

Definition 5.3.14 (Minsky Machine, [Minsky, 1961])**.** *A Minsky machine is a tuple $M = \langle R, Z, z_0, L\rangle$, where*

- *R is a finite set of registers (holding natural numbers)*
- *Z is a finite set of locations*
- *$z_0 \in Z$ is an initial location*
- *$L : Z \to \mathsf{CounterInstr}$ labels each location with an instruction, where*

$$\begin{aligned} \mathsf{CounterInstr} ::= {} & inc(r, z) && \text{increment } r \text{ \& jump to } z \\ | \; & jzd(r, t, f) && \text{if } r = 0 \text{ jump to } t\text{; else decrement } r \text{ \& jump to } f \\ | \; & halt && \text{stop execution} \end{aligned}$$

Fix a Minsky machine $M = \langle R, Z, z_0, L\rangle$. Define a PA $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ as follows:

- The vocabulary $\langle Q, ar\rangle$ is defined by:
  - Each location $z \in Z$ is a nullary predicate symbol.
  - For each register $r \in R$, there are two monadic predicate symbols bot_r and top_r, and one dyadic predicate symbol ln_r.
  - There is one distinguished nullary predicate symbol *init*.
- The labels Σ consist of the M-instructions, a set of "barred" instructions that represent taking the "else" branch of a jzd instruction, and a designated initialization instruction `init`:

$$\Sigma \triangleq \{L(z) : z \in Z\} \cup \{\overline{jzd(r, t, f)} : \exists z.L(z) = jzd(r, t, f)\} \cup \{\texttt{init}\}$$

- The set of indices $N = \mathbb{N}$ is the set of naturals.
- The initial formula $\varphi_{\mathsf{init}} \triangleq init$ requires that the proposition *init* holds in the initial configuration.
- The set of accepting predicates $F \triangleq \emptyset$ is empty. (There are no accepting predicate symbols: the automaton accepts when it transitions to *true*.)
- The transition function δ will be defined in the following.

We design A so that the transition system of A is bisimilar to that of M. A state of M consists of a tuple $\langle z, v\rangle$ where $z \in Z$ is a control state and $v : R \to \mathbb{N}$ is a valuation mapping each register to a natural. Define a relation $\sim$ so that $\langle z, v\rangle \sim \mathcal{C}$ if and only if there exists distinct naturals $\mathtt{i}_{r,0}, ..., \mathtt{i}_{r,v(r)}$ for each register r such that

$$\mathcal{C} = \{z\} \cup \bigcup_{r \in R} \left(\{bot_r(\mathtt{i}_{r,0}), ln_r(\mathtt{i}_{r,0}, \mathtt{i}_{r,1}), ..., ln_r(\mathtt{i}_{r,v(r)-1}, \mathtt{i}_{r,v(r)}), top_r(\mathtt{i}_{r,v(r)_r})\}\right)$$

Thus, the value of a register r is encoded in an A-configuration by a chain of indices in the dyadic predicate ln_r, where the end points are demarcated by the monadic predicates bot_r and top_r. Our task now is to design the transition relation of A so that:

1. (Recalling that $\{init\}$ is the initial configuration of A and $\langle z_0, \lambda r.0\rangle$ is the initial state of M) for all indices $\mathtt{i}$ and configurations $\mathcal{C}$ such that $\{init\} \xrightarrow{\mathtt{init:i}} \mathcal{C}$, we have $\langle z_0, \lambda r.0\rangle \sim \mathcal{C}$.

2. For all $\langle z, v\rangle \sim \mathcal{C}$, if M may transition from $\langle z, v\rangle$ to $\langle z', v'\rangle$, then there exists a letter $\langle \sigma : \mathtt{i}\rangle \in \Sigma(\mathbb{N})$ and a configuration $\mathcal{C}'$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$ and $\langle z', v'\rangle \sim \mathcal{C}'$.

3. For all $\langle z, v\rangle \sim \mathcal{C}$, and for all $\langle \sigma : \mathtt{i}\rangle \in \Sigma(\mathbb{N})$ and configurations $\mathcal{C}'$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, either M halts at $\langle z, v\rangle$ and $\mathcal{C}'$ is accepting or there exists a state $\langle z', v'\rangle$ of M such that M may transition from $\langle z, v\rangle$ to $\langle z', v'\rangle$ and $\langle z', v'\rangle \sim \mathcal{C}'$.

The first condition is obtained by defining

$$\delta(init, \mathtt{init} : i_0) \triangleq z_0 \wedge \bigwedge_{r\in R} \left(bot_r(i_0) \wedge top_r(i_0)\right) .$$

We may define $\delta(init, \sigma : i_0) \triangleq false$ for all $\sigma \neq \mathtt{init}$ to ensure that every word that is accepted by A begins with $\mathtt{init}$.

The second two conditions can be understood in two parts: simulating the control state of the counter machine, and simulating the register state. Simulating the control state of the counter machine is straight-forward, using the nullary predicates corresponding to the control states of M. For each $z \in Z$, define:

$$\delta(z, instr : i_0) \triangleq \begin{cases} z' & \text{if } instr \text{ is } L(z) = inc(r, z') \\ t & \text{if } instr \text{ is } L(z) = jzd(r, t, f) \\ f & \text{if } instr \text{ is } \overline{jzd(r, t, f)} \text{ and } L(z) = jzd(r, t, f) \\ true & \text{if } instr \text{ is } L(z) = halt \\ false & \text{otherwise} \end{cases}$$

Let $r \in R$ be a register. Recall that the state of r is simulated by a chain of the form

$$\{bot_r(\mathtt{i}_{r,0}), ln_r(\mathtt{i}_{r,0}, \mathtt{i}_{r,1}), ..., ln_r(\mathtt{i}_{r,n-1}, \mathtt{i}_{r,n}), top_r(\mathtt{i}_{r,n})\} .$$

To increment register r, we must select a fresh index $\mathtt{i}$ not belonging to $\mathtt{i}_{r,0}, ..., \mathtt{i}_{r,n}$ and add a link $ln(\mathtt{i}_{r,n}, \mathtt{i})$ and set the top index top_r to $\mathtt{i}$:

$$\begin{aligned} \delta(bot_r(i_1), inc(r, z) : i_0) &= bot_r(i_1) \\ \delta(top_r(i_1), inc(r, z) : i_0) &= i_0 \neq i_1 \wedge ln_r(i_1, i_0) \wedge top_r(i_0) \\ \delta(ln_r(i_1, i_2), inc(r, z) : i_0) &= i_0 \neq i_1 \wedge ln_r(i_1, i_2) \end{aligned}$$

To decrement register r, we choose the top index $\mathtt{i}_{r,n}$, ensure that the register is non-empty ($\mathtt{i}_{r,0} \neq$

$\mathtt{i}_{r,n}$), delete the link $ln(\mathtt{i}_{r,n-1}, \mathtt{i}_{r,n})$, and set the top to $\mathtt{i}_{r,n-1}$:

$$\begin{aligned}
\delta(bot_r(i_1), \overline{jzd(r,t,f)} : i_0) &= i_0 \neq i_1 \wedge bot_r(i_1) \\
\delta(top_r(i_1), \overline{jzd(r,t,f)} : i_0) &= i_0 = i_1 \\
\delta(ln_r(i_1,i_2), \overline{jzd(r,t,f)} : i_0) &= (i_2 = i_0 \wedge top_r(i_1)) \vee (i_2 \neq i_0 \wedge ln_r(i_1,i_2))
\end{aligned}$$

To zero-test register r, we simply ensure that there are no ln_r propositions in the configuration:

$$\begin{aligned}
\delta(bot_r(i_1), jzd(r,t,f) : i_0) &= bot_r(i_1) \\
\delta(top_r(i_1), jzd(r,t,f) : i_0) &= top_r(i_1) \\
\delta(ln_r(i_1,i_2), jzd(r,t,f) : i_0) &= false
\end{aligned}$$

On a halt instruction, we transition to *true* so that A accepts:

$$\begin{aligned}
\delta(bot_r(i_1), halt : i_0) &= true \\
\delta(top_r(i_1), halt : i_0) &= true \\
\delta(ln_r(i_1,2), halt : i_0) &= true
\end{aligned}$$

Finally, for any register r, any instruction *instr* that refers to a register other than r is a no-op with respect to the predicate symbols related to r:

$$\begin{aligned}
\delta(bot_r(i_1), instr : i_0) &\triangleq bot_r(i_1) \\
\delta(top_r(i_1), instr : i_0) &\triangleq top_r(i_1) \\
\delta(ln_r(i_1,i_2), instr : i_0) &\triangleq ln_r(i_1,i_2)
\end{aligned}$$

□

Checking that $\sim$ is a bisimulation relation is straight forward.

Monadic predicate automata. Since PA emptiness is undecidable in general, it is interesting to consider sub-classes where it is decidable. We say that a predicate automaton $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ is *monadic* if the arity of each predicate $q \in Q$ is either 0 or 1. We have the following:

Proposition 5.3.15. *Algorithm 5 terminates for monadic predicate automata (i.e., emptiness is decidable for the class of monadic predicate automata, and Algorithm 5 is a decision procedure).*

Proof. A sufficient (but not necessary) condition for Algorithm 5 to terminate is that $\preceq$ is a well-quasi order: that is, for any infinite sequence $\mathcal{C}_1\mathcal{C}_2...$ of configurations there exists $m < n$ such that $\mathcal{C}_m \preceq \mathcal{C}_n$. This is the essential idea behind well-structured transition systems [Finkel, 1987, Abdulla et al., 1996, Finkel and Schnoebelen, 2001]. The search space constructed by Algorithm 5 forms a finitely-branching tree, so by König's lemma [König, 1927], if the tree is infinite (that is, Algorithm 5 does not terminate), then there is an infinite branch. The infinite branch is labelled with an infinite sequence of configurations $\mathcal{C}_1\mathcal{C}_2....$ If $\preceq$ is a well-quasi order, then there must exist some $m < n$ such that $\mathcal{C}_m \preceq \mathcal{C}_n$; but in this case, Algorithm 5 does not expand $\mathcal{C}_n$, contradicting the fact that $\mathcal{C}_1\mathcal{C}_2...$ is an infinite branch.

We will prove that $\preceq$ is a well-quasi ordering in the following. The fact that Algorithm 5 is a decision procedure for the emptiness problem for monadic predicate automata follows from partial correctness (Theorem 5.3.11) and the fact that it terminates.

Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a monadic PA. Let $\mathcal{C}_1\mathcal{C}_2, ...$ be an infinite sequence of configurations. We must show that there exists some m and n such that $m < n$ and $\mathcal{C}_m \preceq \mathcal{C}_n$.

Let Q_0 be the set of nullary predicate symbols of A, and let Q_1 be the set of monadic predicate symbols. Since the set of nullary predicate symbols Q_0 is finite, there is some subset $P \subseteq Q_0$ such that $\mathcal{C}_n \cap Q_0 = P$ for infinitely many n. Let $k_1 < k_2 < k_3 ...$ be an infinite ascending sequence of naturals such that $\mathcal{C}_{k_i} \cap Q_0 = P$ for all i.

Within each configuration $\mathcal{C}_j$, each index $\mathtt{i} \in N$ is associated with some set of monadic predicates $sig_j(\mathtt{i}) \triangleq \{q \in Q_1 : q(\mathtt{i}) \in \mathcal{C}_j\}$. We define a function $f_j : 2^{Q_1} \to \mathbb{N}$ that maps each non-empty set of monadic predicate symbols $P \subseteq Q_1$ to the number of indices associated with P: $f_j(P) \triangleq |\{\mathtt{i} \in N : sig_j(\mathtt{i}) = P\}|$. By Dickson's lemma [Dickson, 1913], the pointwise order on $2^{Q_1} \to \mathbb{N}$ is a well-quasi order, so there exists $m < n$ such that for all $P \subseteq Q_1$, $f_{k_m}(P) \leq f_{k_n}(P)$.

Let $P \subseteq Q_1$ be a non-empty set of monadic predicates. Since $f_{k_m}(P) \leq f_{k_n}(P)$, there exists an injection

$$\alpha_P : \{\mathtt{i} \in N : sig_{k_n}(\mathtt{i}) = P\} \to \{\mathtt{i} \in N : sig_{k_m}(\mathtt{i}) = P\} \ .$$

Define a function $\alpha : \mathsf{support}(\mathcal{C}_{k_n}) \to N$ as $\alpha(\mathtt{i}) \triangleq \alpha_{sig(\mathtt{i})}(\mathtt{i})$.

Since α is injective, the sets $N \setminus \mathsf{support}(\mathcal{C}_{k_n})$ and $N \setminus \alpha(\mathsf{support}(\mathcal{C}_{k_n}))$ have the same cardinality, so there is a bijection β between them. Finally, we define

$$\pi(\mathtt{i}) \triangleq \begin{cases} \alpha(\mathtt{i}) & \text{if } \mathtt{i} \in \mathsf{support}(\mathcal{C}_{k_n}) \\ \beta(\mathtt{i}) & \text{otherwise} \end{cases}$$

We then check that π is an embedding of $\mathcal{C}_{k_n}$ into $\mathcal{C}_{k_m}$, so that $\mathcal{C}_{k_n} \preceq \mathcal{C}_{k_m}$. This holds because:

- By the construction of the sequence $\mathcal{C}_{k_1}\mathcal{C}_{k_2}...$, we have that $\mathcal{C}_{k_m} \cap Q_0 = \mathcal{C}_{k_m} \cap Q_0$, so every nullary proposition of $\mathcal{C}_{k_m}$ belongs to $\mathcal{C}_{k_n}$.
- For any monadic proposition $q(\mathtt{i}) \in \mathcal{C}_{k_m}$, we have $\mathtt{i} \in \mathsf{support}(\mathcal{C}_{k_m})$ so that $\pi(\mathtt{i}) = \alpha_{sig(\mathtt{i})}(\mathtt{i})$, and thus (by definition of $\alpha_{sig(\mathtt{i})}$) $sig_{k_m}(\mathtt{i}) = sig_{k_n}(\pi(\mathtt{i}))$. Expanding the definition of sig, we have

  $$\{q \in Q_1 : q(\mathtt{i}) \in \mathcal{C}_{k_m}\} = \{q \in Q_1 : q(\pi(\mathtt{i})) \in \mathcal{C}_{k_n}\}$$

  and thus $q(\pi(\mathtt{i})) \in \mathcal{C}_{k_n}$. □

The PA $\mathcal{A}(P, \ell_{\mathsf{err}})$ that recognizes the set of error traces of a program P leading to an error location ℓ_{err} is always monadic. However, it is not always the case that the predicate automaton corresponding to a regular proof space is monadic. We say that a regular proof space $\mathscr{H}$ with basis H is monadic if $\mathcal{A}(H)$ is a monadic predicate automaton.

From the construction of $\mathcal{A}(H)$, we observe that the predicate automaton $\mathcal{A}(H)$ is monadic exactly when the set of assertions that appear in H refer to the local variables of at most one thread. That is, a proof space *fails* to be monadic when it makes use of assertions that relate the local variables of two or more threads together (for example, an assertion of the form $x(1) \leq y(2)$). Thus, monadic proof spaces may naturally be compared with thread-modular proofs [Flanagan et al., 2002], which also disallow such assertions. It is easy to show that if there exists a thread-modular proof of some non-reachability property of a program, then there exists a monadic proof space (cf. the iDFG construction in Section 4.4). This correspondence is not exact, however: monadic proof spaces are strictly more

powerful than thread-modular proofs. In particular, one can show that if P is a deterministic Boolean program such that some designated error location ℓ_{err} is unreachable, then there exists a monadic proof space that proves ℓ_{err} is unreachable. To see why, consider that each transition of a deterministic Boolean program can be captured precisely be a finite set of Hoare triples by case-splitting on the values of the variables involved.

Discussion: decidability beyond monadic predicate automata. Note that the reverse of Proposition 5.3.15 is not true, i.e., non-monadic predicate automata do not necessarily cause Algorithm 5 to diverge. For example, the proof space of the thread pooling program from Section 5.2.1 is not monadic (for example, the assertion $\mathtt{end}(1) \leq \mathtt{c}(2)$ is dyadic). And yet, Algorithm 5 terminates for this example. We will informally discuss some other classes that generalize the monadic condition for which Algorithm 5 terminates.

One such generalization is *effectively monadic* PA, where there is a finite set of indices $D \subseteq N$ such that in any reachable minimal (with respect to $\preceq$) configuration $\mathcal{C}$, for all $q(\mathtt{i}_1, ..., \mathtt{i}_n) \in \mathcal{C}$ we have at most one of $\mathtt{i}_1, ..., \mathtt{i}_n$ *not* in D. Intuitively, effectively monadic PAs can be used to reason about programs where one or more processes play a distinguished role (e.g., a client/server program, where there is a single distinguished server but arbitrarily many clients).

Another alternative is *boundedly non-monadic* PA, where there exists some bound K such that in any reachable minimal (with respect to $\preceq$) configuration $\mathcal{C}$, the cardinality of the set $\{q(\mathtt{i}_1, ..., \mathtt{i}_n) \in \mathcal{C} : ar(q) > 1\}$ is less than K. The thread pooling example from Section 5.2.1 is boundedly non-monadic with a bound of 3. Intuitively, boundedly non-monadic PAs can be used to reason about programs that do not require "unbounded chains" of inter-thread relationships.

5.3.6 Certificates for Emptiness

Since predicate automata emptiness is not decidable in the general case, how can one be convinced that a predicate automaton accepts an empty language? This section develops *emptiness certificates*, which serve as formal proofs of emptiness. The key requirements of emptiness certificates is that they should be *sound* in the sense that the existence of an emptiness certificate for an automaton implies that it accepts an empty language,[3] and that it should be possible to mechanically verify that an emptiness certificate is valid.

An emptiness certificate is simply an inductive invariant for the predicate automaton that shows that every configuration reachable from an initial configuration is not accepting.

Definition 5.3.16. *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. An* emptiness certificate *for A is an existential sentence of the form $\varphi = \exists i_0, ..., i_n.\psi$ where $\psi \in \mathcal{F}(Q, ar)$ is a positive formula over the vocabulary of A, and such that:*

1. *$\varphi_{init} \models \varphi$,*
2. *for all $\mathcal{C}, \mathcal{C}', \sigma, \mathtt{i}$ such that $\mathcal{C} \models \varphi$ and $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, we have $\mathcal{C}' \models \varphi$, and*
3. *every model of φ is rejecting.*

[3] Note that any notion of certificate is necessarily *incomplete* (in the sense that emptiness of an automaton does not imply existence of an emptiness certificate) because PA emptiness is not co-recursively enumerable.

The following proposition states that emptiness certificates are sound in the sense that existence of an emptiness certificate for an automaton implies that it accepts an empty language.

Proposition 5.3.17. *Let A be a PA such that there exists an emptiness certificate for A. Then $\mathcal{L}(A) = \emptyset$.*

Besides soundness, the other important property of emptiness certificates is that they can be checked mechanically. The following proposition shows that this is the case.

Proposition 5.3.18. *Let A be a predicate automaton, and let φ be a positive existential sentence over the vocabulary of A. There is a decision procedure to determine whether φ is an emptiness certificate for A.*

Proof. First, we define a *symbolic* transition function. Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. Define a function $\hat{\delta}$ that maps each existential sentence over the vocabulary of A and each label in Σ to an existential sentence over the vocabulary of A as follows:

$$\hat{\delta}(\exists i_0, ..., i_n.\varphi, \sigma) \triangleq \exists i_0, ..., i_{n+1}.\hat{\delta}^n(\varphi, \sigma)$$

where

$$\begin{aligned}
\hat{\delta}^n(q(i_{j_1}, ..., i_{j_{ar(q)}}), \sigma) &\triangleq \delta(q, \sigma)[i_0 \mapsto i_{n+1}, i_1 \mapsto i_{j_1}, ..., i_{ar(q)} \mapsto i_{j_{ar(q)}}] \\
\hat{\delta}^n(i = j, \sigma) &\triangleq i = j \\
\hat{\delta}^n(i \neq j, \sigma) &\triangleq i \neq j \\
\hat{\delta}^n(\varphi \wedge \psi, \sigma) &\triangleq \hat{\delta}^n(\varphi, \sigma) \wedge \hat{\delta}^n(\psi, \sigma) \\
\hat{\delta}^n(\varphi \vee \psi, \sigma) &\triangleq \hat{\delta}^n(\varphi, \sigma) \vee \hat{\delta}^n(\psi, \sigma)
\end{aligned}$$

The symbolic transition function $\hat{\delta}$ lifts the transition function δ from propositions to existential sentences. The following lemma confirms that the lifting is sound.

Lemma 5.3.19. *Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA, let φ be a positive existential sentence over the vocabulary of A, let $\mathcal{C}, \mathcal{C}'$ be configurations of A such that $\mathcal{C} \models \varphi$, and $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$ for some $\sigma \in \Sigma$ and $\mathtt{i} \in N$. Then $\mathcal{C}' \models \hat{\delta}(\varphi, \sigma)$.*

Proof. Let $A = \langle Q, ar, \Sigma, N, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA, and let $\varphi = \exists i_0, ..., i_n.\psi$, with $\psi \in \mathcal{F}(Q, ar)$. Let $\mathcal{C}, \mathcal{C}'$ be configurations of A, and let $\langle \sigma : \mathtt{i}\rangle \in \Sigma(N)$ such that $\mathcal{C} \models \varphi$, and $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$.

Since $\mathcal{C} \models \varphi$, there exists a valuation $\mu : \{i_0, ..., i_n\} \to N$ such that $\mathcal{C}, \mu \models \psi$. We prove that $\mathcal{C}', \mu[i_{n+1} \leftarrow \mathtt{i}] \models \hat{\delta}^n(\psi, \sigma)$ by induction on ψ:

- Base case $q(i_{j_1}, ...i_{j_{ar(q)}})$. Since $\mathcal{C}, \mu \models q(i_{j_1}, ..., i_{j_{ar(q)}})$, we must have $q(\mu(i_{j_1}), ..., \mu(i_{j_{ar(q)}})) \in \mathcal{C}$. Since $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, we must have

  $$\mathcal{C}' \models \delta(q, \sigma)[i_0 \mapsto \mathtt{i}, i_1 \mapsto \mu(i_{j_1}), ..., i_{ar(q)} \mapsto \mu(i_{j_{ar(q)}})]$$

  and thus

  $$\begin{aligned}
  \mathcal{C}', \mu' &\models \delta(q, \sigma)[i_0 \mapsto i_{n+1}, i_1 \mapsto i_{j_1}, ..., i_{ar(q)} \mapsto i_{j_{ar(q)}}] \\
  &\equiv \hat{\delta}^n(q(i_{j_1}, ...i_{j_{ar(q)}}), \sigma)\ .
  \end{aligned}$$

- Base case $i = j$. Since $\mathcal{C}, \mu \models i = j$, we must have $\mu(i) = \mu(j)$. It follows that $\mu'(i) = \mu'(j)$, and thus
$$\mathcal{C}', \mu' \models i = j \equiv \hat{\delta}^n(i = j, \sigma)$$
- Base case $i \neq j$ (similar to $i = j$). Since $\mathcal{C}, \mu \models i \neq j$, we must have $\mu(i) \neq \mu(j)$. It follows that $\mu'(i) \neq \mu'(j)$, and thus
$$\mathcal{C}', \mu' \models i \neq j \equiv \hat{\delta}^n(i \neq j, \sigma)$$
- Induction step $\varphi \wedge \psi$. Since $\mathcal{C}, \mu \models \varphi \wedge \psi$, we must have $\mathcal{C}, \mu \models \varphi$ and $\mathcal{C}, \mu \models \psi$. By the induction hypothesis, we have $\mathcal{C}', \mu' \models \hat{\delta}^n(\varphi, \sigma)$ and $\mathcal{C}', \mu' \models \hat{\delta}^n(\psi, \sigma)$, and thus
$$\mathcal{C}', \mu' \models \hat{\delta}^n(\varphi, \sigma) \wedge \hat{\delta}^n(\psi, \sigma) \equiv \hat{\delta}^n(\varphi \wedge \psi, \sigma)$$
- Induction step $\varphi \vee \psi$ (similar to $\varphi \wedge \psi$). Since $\mathcal{C}, \mu \models \varphi \vee \psi$, we must have $\mathcal{C}, \mu \models \varphi$ or $\mathcal{C}, \mu \models \psi$. By the induction hypothesis, we have $\mathcal{C}', \mu' \models \hat{\delta}^n(\varphi, \sigma)$ or $\mathcal{C}', \mu' \models \hat{\delta}^n(\psi, \sigma)$, and thus
$$\mathcal{C}', \mu' \models \hat{\delta}^n(\varphi, \sigma) \vee \hat{\delta}^n(\psi, \sigma) \equiv \hat{\delta}^n(\varphi \vee \psi, \sigma)$$ ■

The proposition comes from the fact that each of the three conditions of emptiness certificates can be reduced to checking the validity of an $\exists^*\forall^*$ formula over the vocabulary of A (i.e., a formula consisting of a sequence of existential quantifiers, a sequence of universal quantifiers, followed by a quantifier-free formula). Checking validity of such formulas (called the "effectively propositional" or "Bernays-Schönfinkel-Ramsey" fragment of first-order logic) is decidable [Ramsey, 1930].

1. $\varphi_{\mathsf{init}} \models \varphi$ holds iff $(\neg\varphi_{\mathsf{init}}) \vee \varphi$ is valid. Since φ_{init} is quantifier-free and φ is existential, $(\neg\varphi_{\mathsf{init}}) \vee \varphi$ is existential.
2. By Lemma 5.3.19, condition 2 holds iff for all σ, we have $\hat{\delta}(\varphi, \sigma) \models \varphi$, or equivalently $\neg\hat{\delta}(\varphi, \sigma) \vee \varphi$ is valid. Since $\hat{\delta}(\varphi, \sigma)$ is an existential sentence, $\neg\hat{\delta}(\varphi, \sigma)$ is equivalent to a universal sentence, and $\neg\hat{\delta}(\varphi, \sigma) \vee \varphi$ is equivalent to an $\exists^*\forall^*$ sentence.
3. Every model of φ is rejecting iff φ entails the formula $\psi_{reject} \triangleq \bigvee_{q \in Q \setminus F} \exists j_1, ..., j_{ar(q)}.q(j_1, ..., j_{ar(q)})$, or equivalently if $\neg\varphi \vee \psi_{reject}$ is valid. Since φ is an existential sentence, $\neg\varphi$ is equivalent to universal sentence, so $\neg\varphi \vee \psi_{reject}$ is equivalent to an $\exists^*\forall^*$ sentence. □

5.4 Completeness

The method of using global inductive invariants to prove correctness of concurrent programs dates back to the seminal work of Ashcroft [Ashcroft, 1975]. Ashcroft's method originally applied to programs with finitely many threads, but can be adapted to the infinite case by admitting into the language of assertions universal quantification over threads. In this section, we prove the completeness of proof spaces relative to Ashcroft's method. This establishes that the simple assertional language of proof spaces combined with the inference rules of Sequencing, Symmetry, and Conjunction make up all the expressive power of the rich assertional language of Ashcroft proofs.

Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N \rangle$ be a program. The syntax of *global assertions* for P, the assertional language of Ashcroft proofs is formalized in Figure 5.5. The logic has three sorts `Int` (integers), `Thread` (thread

$$\textsc{TInt}\ \frac{n \in \mathbb{Z}}{n : \mathsf{Term}_{\mathtt{Int}}} \qquad \textsc{TGlobal}\ \frac{x \in \mathsf{GV}}{x : \mathsf{Term}_{\mathtt{Int}}} \qquad \textsc{TLocal}\ \frac{y \in \mathsf{LV} \qquad t \in \mathsf{Term}_{\mathtt{Thread}}}{y(t) : \mathsf{Term}_{\mathtt{Int}}} \qquad \textsc{TThread}\ \frac{i \in \mathsf{TV}}{i : \mathsf{Term}_{\mathtt{Thread}}} \qquad \textsc{TThreadLoc}\ \frac{t : \mathsf{Term}_{\mathtt{Thread}}}{loc(t) : \mathsf{Term}_{\mathtt{Loc}}(L)}$$

$$\textsc{TLoc}\ \frac{\ell \in L}{\ell : \mathsf{Term}_{\mathtt{Loc}}(L)} \qquad \textsc{TAdd}\ \frac{t_1 : \mathsf{Term}_{\mathtt{Int}} \qquad t_2 : \mathsf{Term}_{\mathtt{Int}}}{t_1 + t_2 : \mathsf{Term}_{\mathtt{Int}}} \qquad \textsc{TMul}\ \frac{n \in \mathbb{Z} \qquad t : \mathsf{Term}_{\mathtt{Int}}}{n \times t : \mathsf{Term}_{\mathtt{Int}}}$$

$$\textsc{FLt}\ \frac{t_1 : \mathsf{Term}_{\mathtt{Int}} \qquad t_2 : \mathsf{Term}_{\mathtt{Int}}}{t_1 < t_2 : \mathsf{Assertion}} \qquad \textsc{FEq}\ \frac{t_1 : \mathsf{Term}_{\alpha} \qquad t_2 : \mathsf{Term}_{\alpha} \qquad \alpha \in \{\mathtt{Int}, \mathtt{Loc}, \mathtt{Thread}\}}{t_1 = t_2 : \mathsf{Assertion}(L)}$$

$$\textsc{FAnd}\ \frac{\varphi : \mathsf{Assertion}(L) \qquad \psi : \mathsf{Assertion}(L)}{\varphi \wedge \psi : \mathsf{Assertion}(L)} \qquad \textsc{FOr}\ \frac{\varphi : \mathsf{Assertion}(L) \qquad \psi : \mathsf{Assertion}(L)}{\varphi \vee \psi : \mathsf{Assertion}(L)}$$

$$\textsc{GAssert}\ \frac{\varphi : \mathsf{Assertion}(L)}{\varphi : \mathsf{GlobalAssertion}(L)} \qquad \textsc{GForall}\ \frac{\Phi : \mathsf{GlobalAssertion}(L)}{\forall i \in \mathtt{Thread}.\Phi : \mathsf{GlobalAssertion}(L)}$$

Figure 5.5: Syntax of global assertions over a set of control locations L

identifiers), and `Loc` (control locations). Each local variable is treated as a function symbol of type `Thread` $\to$ `Int` (i.e, the interpretation of a local variable x is a function mapping each thread `i` to the value of `i`'s copy of x). The only terms of sort `Thread` are variables drawn from a syntactic category TV of thread variable symbols. We typically use $i, j, i_1, i_2 \ldots$ to refer to thread variables. Each location $\ell \in \mathsf{Loc}$ is treated as a `Loc`-sorted constant symbol, and additionally there is a function symbol *loc* of sort `Thread` $\to$ `Loc` that maps each thread to its control location. The models of interest for global assertions are the states $\langle \rho, loc \rangle$ of P (so, for example, when we say that one assertion Φ entails another Ψ, we mean that every state of P that satisfies Φ also satisfies Ψ).

Recall that for any instruction *instr*, $\mathsf{Formula}[\![instr]\!]$ is a transition formula (over primed and unprimed variables) representing how *instr* changes the state of the thread that executes it. For any command $\sigma = \langle \ell, instr, \ell' \rangle$, we define a *global* formula $\mathsf{GlobalFormula}[\![\sigma]\!]$ that represents how the execution of σ by *some* thread affects the global state of the program:

$$\mathsf{GlobalFormula}[\![\sigma]\!] \triangleq \exists i \in \mathtt{Thread}.\mathsf{step}(\sigma, i) \wedge \forall j \in \mathtt{Thread}.j = i \vee \mathsf{nop}(j)$$

where $\mathsf{step}(\sigma, i)$ captures the effect of σ on the globals and the locals of thread i, and $\mathsf{nop}(j)$ captures the *lack* of effect of σ on the locals of thread j:

$$\mathsf{step}(\sigma, i) \triangleq loc(i) = \ell \wedge loc'(i) = \ell' \wedge \mathsf{Formula}[\![instr]\!][x \mapsto x(i)]_{x \in \mathsf{LV}}$$
$$\mathsf{nop}(j) \triangleq loc(j) = loc'(j) \wedge \bigwedge_{x \in \mathsf{LV}} x(j) = x'(j)$$

Above $[x \mapsto x(i)]_{x \in \mathsf{LV}}$ denotes the substitution that replaces every local variable x with $x(i)$. Note that the Hoare triple $\{\varphi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\psi\}$ is valid if and only if $\varphi \wedge \mathsf{step}(\sigma, \mathtt{i}) \wedge \forall j \in \mathtt{Thread}.j = \mathtt{i} \vee \mathsf{nop}(j)$ entails ψ', where ψ' is the formula obtained from ψ be replacing all symbols with their primed counterparts.

We may now formally define Ashcroft proofs:

Definition 5.4.1 (Ashcroft proof)**.** *Given a program $P = \langle \mathsf{Loc}, \Sigma, \ell_{init}, N\rangle$ and an error location ℓ_{err}, an* Ashcroft proof *is a global assertion Φ_{inv} such that*

1. Initiation: $\forall i \in \mathit{Thread}.loc(i) = \ell_{init}$ *entails* Φ_{inv},
2. Consecution: *For every command $\sigma \in \Sigma$, we have that $\Phi_{inv} \wedge \mathit{GlobalFormula}[\![\sigma]\!]$ entails Φ'_{inv} (where Φ'_{inv} denotes the formula obtained by replacing the symbols in Φ_{inv} with their primed copies), and*
3. Safety: Φ_{inv} *entails* $\forall i \in \mathit{Thread}.loc(i) \neq \ell_{err}$.

Clearly, the existence of an Ashcroft proof for a program and error location implies that the error location is unreachable.

Ashcroft proofs are defined over a rich assertional language that includes control assertions and universal quantification over thread variables. These features are typical of program logics for concurrent programs with unbounded parallelism. The following theorem states that these "exotic" features are not required in the setting of proof spaces: any Ashcroft proof can be translated into a proof space that does not use them.

Theorem 5.4.2 (Relative completeness)**.** *Let P be a program, and let ℓ_{err} be an error location. If there is an Ashcroft proof that ℓ_{err} is unreachable in P, then there is a regular proof space $\mathscr{H}$ such that for all error traces $\tau \in \mathcal{L}(P, \ell_{err})$, the Hoare triple $\{true\}\ \tau\ \{false\}$ belongs to $\mathscr{H}$.*

Proof. To simplify notation, we will prove the result only for the case that the Ashcroft proof has two quantified thread variables. The generalization to an arbitrary number of quantified thread variables is straightforward.

Let $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, N\rangle$ be a program and let Φ_{inv} be an Ashcroft proof. Without loss of generality, we may assume that Φ_{inv} is written in the form

$$\begin{aligned}\Phi_{\mathsf{inv}} \equiv &\big(\forall i \in \texttt{Thread}.\ \bigvee_{\ell \in \mathsf{Loc}} loc(i) = \ell \wedge \varphi_\ell(i)\big)\\ &\wedge \big(\forall i, j \in \texttt{Thread}. i = j \vee \bigvee_{\ell,\ell' \in \mathsf{Loc}\times\mathsf{Loc}} loc(i) = \ell \wedge loc(j) = \ell' \wedge \varphi_{\ell,\ell'}(i,j)\big)\big)\end{aligned}$$

where each $\varphi_\ell(i)$ and $\varphi_{\ell,\ell'}(i,j)$ is a linear integer arithmetic formula (i.e., does not contain *loc*). Intuitively, writing Φ_{inv} in this form simply partitions the formula according to the control state of each thread and whether the two quantified thread variables are equal (corresponding to the first conjunct of Φ_{inv}) or not (the second). Making these case distinctions explicit in the formula simplifies the process of extracting Hoare triples for the proof space. Further, we make the following assumptions (again, without loss of generality):

(i) For all locations ℓ and ℓ', $\varphi_{\ell,\ell'}(i,j)$ is syntactically equal to $\varphi_{\ell',\ell}(j,i)$ (observe that replacing the occurrence of $\varphi_{\ell,\ell'}(i,j)$ in Φ_{inv} with $\varphi_{\ell,\ell'}(i,j) \wedge \varphi_{\ell',\ell}(j,i)$ results in an equivalent formula).

(ii) For all locations ℓ, $\varphi_{\ell,\ell'}(i,j)$ entails $\varphi_\ell(i)$ and $\varphi'_\ell(j)$ (observe that replacing the occurrence of $\varphi_{\ell,\ell'}(i,j)$ in Φ_{inv} with $\varphi_{\ell,\ell'}(i,j) \wedge \varphi_\ell(i) \wedge \varphi_\ell(j)$ results in an equivalent formula).

(iii) $\varphi_{\ell_{\mathsf{init}}}(i)$ and $\varphi_{\ell_{\mathsf{init}},\ell_{\mathsf{init}}}(i,j)$ are both *true* (cf. the initiation condition of Ashcroft proofs),

(iv) $\varphi_{\ell_{\mathsf{err}}}(i)$ is *false* (cf. the safety condition of Ashcroft proofs), and

(v) For every $\ell_1, \ell_2 \in \mathsf{Loc}$, the assertion $\varphi_{\ell_1}(i)$ refers to a local variable of thread i and the assertion $\varphi_{\ell_1,\ell_2}(i)$ refers to local variables of both i and j (observe that, for any local variable x, $\varphi_{\ell_1}(i)$ is equivalent to $\varphi_{\ell_1}(i) \wedge x(i) = x(i)$, and $\varphi_{\ell_1,\ell_2}(i,j)$ is equivalent to $\varphi_{\ell_1,\ell_2}(i,j) \wedge x(i) = x(i) \wedge x(j) = x(j)$).

Let $\mathtt{i}, \mathtt{j}, \mathtt{k} \in N$ be distinct threads. We construct a set of basic Hoare triples H as follows:

$$
\begin{aligned}
H \triangleq\ & \{\{\varphi_{\ell_1}(\mathtt{i})\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi_{\ell_1'}(\mathtt{i})\} : \sigma = \langle \ell_1, \mathit{instr}, \ell_1'\rangle \in \Sigma\} \\
& \cup \{\{\varphi_{\ell_1,\ell_2}(\mathtt{i})\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi_{\ell_2}(\mathtt{i})\} : \sigma = \langle \ell_1, \mathit{instr}, \ell_1'\rangle \in \Sigma, \ell_2 \in \mathsf{Loc}\} \\
& \cup \{\{\varphi_{\ell_1,\ell_2}(\mathtt{i})\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi_{\ell_1',\ell_2}(\mathtt{i})\} : \sigma = \langle \ell_1, \mathit{instr}, \ell_1'\rangle \in \Sigma, \ell_2 \in \mathsf{Loc}\} \\
& \cup \{\{\varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j}) \wedge \varphi_{\ell_1,\ell_3}(\mathtt{i},\mathtt{k}) \wedge \varphi_{\ell_2,\ell_3}(\mathtt{j},\mathtt{k})\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi_{\ell_2,\ell_3}(\mathtt{j},\mathtt{k})\} : \sigma = \langle \ell_1, \mathit{instr}, \ell_1'\rangle \in \Sigma, \ell_2, \ell_3 \in \mathsf{Loc}\}
\end{aligned}
$$

We now must show that (1) each Hoare triple in H is basic (i.e., satisfies Definition 5.2.2) and (2) for every error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$, $\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$ belongs to the proof space generated by H.

We begin by proving (1). It is easily observed that every Hoare triple in H has a singleton post-condition. By assumption (v), condition 2 of Definition 5.2.2 holds as well. It remains to show that each Hoare triple in H is valid. We will prove only the validity of the Hoare triple

$$\{\varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j})\}\ \langle\sigma : \mathtt{i}\rangle\ \{\varphi_{\ell_1',\ell_2}(\mathtt{i},\mathtt{j})\}$$

where $\mathsf{src}(\sigma) = \ell_1$ and $\mathsf{tgt}(\sigma) = \ell_1'$. The other cases are similar.

Let $\langle \rho, \mathit{loc}\rangle$ be a state of P such that $\langle \rho, \mathit{loc}\rangle \models \varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j})$, and let $\langle \rho, \mathit{loc}\rangle \xrightarrow{\langle\sigma:\mathtt{i}\rangle} \langle \rho', \mathit{loc}'\rangle$. We must show that $\langle \rho', \mathit{loc}'\rangle \models \varphi_{\ell_1',\ell_2}(\mathtt{i},\mathtt{j})$.

Consider the program $P_* \triangleq \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, \{\mathtt{i},\mathtt{j}\}\rangle$ that is the same as P except that it has only two threads, $\mathtt{i}$ and $\mathtt{j}$. Define ρ_* to the store of $\widehat{P}$ obtained by restricting the domain of ρ to $\mathsf{GV} \cup \mathsf{LV}(\{\mathtt{i},\mathtt{j}\})$, the globals and locals of threads $\mathtt{i}$ and $\mathtt{j}$. Define ρ_*' similarly. Define loc_* to be the control state (of P_*) that maps $\mathtt{i} \mapsto \ell_1$ and $\mathtt{j} \mapsto \ell_2$, and let loc_*' be be the control state that maps $\mathtt{i} \mapsto \ell_1'$ and $\mathtt{j} \mapsto \ell_2$.

Observe that since $\langle \rho, \mathit{loc}\rangle \models \varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j})$ and the states $\langle \rho, \mathit{loc}\rangle$ and $\langle \rho_*, \mathit{loc}_*\rangle$ agree on the interpretation of every symbol in $\varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j})$, we have $\langle \rho_*, \mathit{loc}_*\rangle \models \varphi_{\ell_1,\ell_2}(\mathtt{i},\mathtt{j})$. By assumption (i) we have $\langle \rho_*, \mathit{loc}_*\rangle \models \varphi_{\ell_2,\ell_1}(\mathtt{j},\mathtt{i})$, and by assumption (ii) we have $\langle \rho_*, \mathit{loc}_*\rangle \models \varphi_{\ell_1}(\mathtt{i})$ and $\langle \rho_*, \mathit{loc}_*\rangle \models \varphi_{\ell_2}(\mathtt{j})$. It follows that (since $\mathtt{i}$ and $\mathtt{j}$ are the only threads of P_*) $\langle \rho_*, \mathit{loc}_*\rangle \models \Phi_{\mathsf{inv}}$. Since $\langle \rho_*, \mathit{loc}_*\rangle$ is a model of Φ_{inv} and $\langle \rho_*, \mathit{loc}_*\rangle \xrightarrow{\langle\sigma:\mathtt{i}\rangle} \langle \rho_*', \mathit{loc}_*'\rangle$, it follows from the consecution condition of Ashcroft proofs that $\langle \rho_*', \mathit{loc}_*'\rangle \models \langle \rho_*', \mathit{loc}_*'\rangle \models \Phi_{\mathsf{inv}}$. Thus

$$\langle \rho_*', \mathit{loc}_*'\rangle \models \bigvee_{\ell,\ell' \in \mathsf{Loc}\times\mathsf{Loc}} \mathit{loc}(\mathtt{i}) = \ell \wedge \mathit{loc}(\mathtt{j}) = \ell' \wedge \varphi_{\ell,\ell'}(\mathtt{i},\mathtt{j})\ .$$

Since $\mathit{loc}_*'(\mathtt{i}) = \ell_1'$ and $\mathit{loc}_*'(\mathtt{j}) = \ell_2$, we must have $\langle \rho_*', \mathit{loc}_*'\rangle \models \varphi_{\ell_1',\ell_2}(\mathtt{i},\mathtt{j})$. Since $\langle \rho', \mathit{loc}'\rangle$ and $\langle \rho_*', \mathit{loc}_*'\rangle$ agree on the interpretation of each symbol appearing in $\varphi_{\ell_1',\ell_2}(\mathtt{i},\mathtt{j})$, we have $\langle \rho', \mathit{loc}'\rangle \models \varphi_{\ell_1',\ell_2}(\mathtt{i},\mathtt{j})$, and we are done.

Having shown that H contains only basic Hoare triples, it remains to show condition (2): for every error trace $\tau \in \mathcal{L}(P, \ell_{\mathsf{err}})$, $\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$ belongs to the proof space generated by H. We will prove this by exhibiting an emptiness certificate (Definition 5.3.16) for the predicate automaton $\mathcal{A}(P, \ell_{\mathsf{err}}) \cap \overline{\mathcal{A}(H)}$.

Recall that the vocabulary of $\mathcal{A}(P, \ell_{\mathsf{err}}) \cap \overline{\mathcal{A}(H)}$ consists of the vocabulary of $\mathcal{A}(P, \ell_{\mathsf{err}})$ along with the "negated" vocabulary of $\mathcal{A}(H)$. That is, the set of predicate symbols is

$$\mathsf{Loc} \cup \{I, \mathsf{err}\} \cup \{\overline{[\varphi_\ell]} : \ell \in \mathsf{Loc}\} \cup \{\overline{[\varphi_{\ell,\ell'}]} : \ell, \ell' \in \mathsf{Loc}\}.$$

For any $\ell \in \mathsf{Loc}$ we use $\overline{\ell(i)}$ as shorthand for

$$\bigvee \{\ell'(i) : \ell' \in \mathsf{Loc} \wedge \ell \neq \ell'\} .$$

The following formula is such an emptiness certificate:

$$\begin{aligned} I \wedge \Big(& \big(\exists i. \bigwedge_{\ell \in \mathsf{Loc}} \overline{\ell(i)} \vee \overline{[\varphi_\ell]}(i)\big) \\ & \vee \big(\exists i, j. i \neq j \wedge \bigwedge_{\ell,\ell' \in \mathsf{Loc}\times\mathsf{Loc}} \overline{\ell(i)} \vee \overline{\ell'(j)} \vee \overline{[\varphi_{\ell,\ell'}]}(i,j)\big)\big) \\ & \vee (\mathsf{err} \wedge \overline{[\mathit{false}]})\Big) \end{aligned}$$

The conditions of Definition 5.3.16 can easily be checked. The intuition behind this emptiness certificate comes from the observation that the negation of a forwards inductive invariant for a transition system (a formula that holds in the initial state, is preserved by the transition relation, and entails some property of interest) is a backwards inductive invariant (a formula that is entailed by the negation of the some property of interest, is preserved by the reversed transition relation, and which does not intersect the initial states). □

5.5 Verification algorithm

In this section, we describe a simple software model checking algorithm based on proof spaces. The algorithm (Algorithm 7) follows the trace abstraction paradigm depicted in Figure 3.6.

The algorithm uses a sub-procedure proof-space (Algorithm 6) to construct proof spaces from traces. Given a trace τ and a post-condition φ such that $\{\mathit{true}\}\ \tau\ \{\varphi\}$ is valid, proof-space(τ, φ) computes a set of basic Hoare triples H that generates a proof space $\mathscr{H}$ such that $\{\mathit{true}\}\ \tau\ \{\varphi\}$ belongs to $\mathscr{H}$. The algorithm is similar to the algorithm `construct-idfg` for constructing iDFG proofs from a trace (Algorithm 3). And similarly to `construct-idfg`, `proof-space` makes use of a sub-procedure Interpolate that, given a trace τ, a command $\langle \sigma : \mathtt{i} \rangle$, and an assertion φ such that the Hoare triple $\{\mathit{true}\}\ \tau\langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$ is valid, computes a set of *intermediate assertions* Ψ such that both

$$\{\mathit{true}\}\ \tau'\ \{\bigwedge \Psi\} \quad \text{and} \quad \{\bigwedge \Psi\}\ \langle \sigma : \mathtt{i} \rangle\ \{\varphi\}$$

are valid Hoare triples. Since we require `proof-space` to compute a set of *basic* Hoare triples (Definition 5.2.2) we additionally require that Interpolate produce intermediate assertions in which only the local variables of relevant threads appear. Craig interpolation [Henzinger et al., 2004b, Craig, 1957] is a natural way to implement Interpolate that guarantees this property. The proof-space procedure essentially just propagates Interpolate backwards across the trace, and collects the resulting basic Hoare triples.

Algorithm 7 takes as input a program P and an error location ℓ_{err} and (if it terminates) returns either

Input : Non-empty trace τ and an assertion φ such that $\{true\}\ \tau\ \{\varphi\}$ is valid
Output: A set of basic Hoare triples H that generates a proof space containing $\{true\}\ \tau\ \{\varphi\}$
Let $\tau'\langle\sigma : \mathtt{i}\rangle = \tau$;
$\Psi \leftarrow \mathsf{Interpolate}(\tau', \langle\sigma : \mathtt{i}\rangle, \varphi)$;
$H \leftarrow \emptyset$;
for $\psi \in \Psi$ **do**
  $H_\psi \leftarrow \mathsf{proof\text{-}space}(\tau', \psi)$;
  $H \leftarrow H \cup H_\psi$;
end
return H

Algorithm 6: $\mathsf{proof\text{-}space}(\tau, \varphi)$

a counter-example τ showing that ℓ_{err} is reachable or a basis H of a proof space that proves that it is not. The algorithm operates by repeatedly sampling error traces τ of P for which $\{true\}\ \tau\ \{false\}$ is not in the proof space generated by H. If τ is feasible then the program is incorrect and the counter-example τ is returned. Otherwise, we add the Hoare triples from $\mathsf{proof\text{-}space}(\tau)$ to H. If we are *unable* to sample a trace τ for which $\{true\}\ \tau\ \{false\}$ is not in the proof space generated by H, then H is a basis for a proof space that proves the correctness of P, and we return H. The high-level properties of this algorithm are summarized in the following theorem.

Theorem 5.5.1. *Algorithm 7 is sound and is complete for counter-examples: given a program P, if Algorithm 7 returns Safe, then ℓ_{err} is unreachable; if ℓ_{err} is reachable, then Algorithm 7 returns a feasible trace that reaches ℓ_{err}.*

Proof. Soundness follows from Theorem 3.3.1 and Theorem 5.3.11. Completeness follows from progress (the same (possibly spurious) counter-example may not be considered twice), completeness of the emptiness check (Theorem 5.3.11), and the fact that Algorithm 5 is a breadth-first search, and therefore always finds a trace in the language of a PA of minimal length. □

$H \leftarrow \emptyset$; `/* Basis for a proof space */`
while $(\mathcal{A}(P, \ell_{err}) \cap \overline{\mathcal{A}(H)}) \neq \emptyset$ **do** `/* Algorithm 5 */`
  Select τ from $\mathcal{A}(P, \ell_{\mathsf{err}}) \cap \overline{\mathcal{A}(H)}$; `/* Algorithm 5 */`
  if τ *is feasible* **then**
    return *Counter-example* τ
  else
    $H \leftarrow H \cup \mathsf{proof\text{-}space}(\tau)$
  end
end
return *Proof* H

Algorithm 7: $\mathrm{Verify}(P, \ell_{\mathsf{err}})$

5.6 Related work

Unbounded parallelism and unbounded memory Thread modular reasoning has been applied in the context of reasoning about unboundedly many infinite-state processes. Thread-modular reasoning sacrifices some proof-theoretic power (in particular, the ability to reason about correlated local

variables of different threads) to obtain more tractable reasoning. [Henzinger et al., 2003] uses a CEGAR approach to synthesize thread modular proofs, where the behaviour of a distinguished thread is approximated from above and the behaviour of environment threads is approximated from below. In [Henzinger et al., 2004a], Henzinger et al. develop a rich environment model for thread-modular reasoning that tracks a finite-state abstraction of each environment thread, using counter abstraction to cope with the fact that there are unbounded many threads. Like monadic proof spaces, [Henzinger et al., 2004a] is complete for reasoning about programs with finite-state threads (but can reason about infinite-state threads as well).

Indexed predicate abstraction allows predicates to have free (thread) variables (e.g., $x(i) \geq 0$), with the ultimate goal of computing an Ashcroft invariant (i.e., a universally quantified invariant) of a fixed quantifier depth. Admitting free thread variables allows indexed predicates to be combined "under the quantifier" and thus compute complex quantified invariants from simple components. In view of the symmetry closure condition of proof spaces, indexed predicates serve a similar function to our ground Hoare triples, and our use of inference rules shares the goal of [Lahiri and Bryant, 2007] to compute complex invariants from simple components. [Lahiri and Bryant, 2007] uses a theorem prover to reason about universal quantifiers and program data simultaneously, and determines an Ashcroft invariant for a program by computing the least fixed point in a finite abstract domain determined by a set of indexed predicates. Our technique does not require a theorem prover that supports universal quantifiers, and the predicate automata inclusion check algorithm replaces the abstract fixed point computation. The separation between reasoning about data and thread quantification provides a fresh perspective on what exactly makes reasoning about unbounded parallelism difficult, and enables us to state and prove results such as the decidability of the monadic case (i.e. Proposition 5.3.15), which has no obvious analogue in the setting of [Lahiri and Bryant, 2007].

Dual reference programs allow two types of predicates: *single-thread predicates* (*mixed assertions* in the terminology in [Donaldson et al., 2012]), which refer to globals and the locals of one thread, and *inter-thread predicates*, which refer to globals and the locals of two threads, where one thread is universally quantified (e.g., $\forall j.x(i) < x(j)$). Although dual reference programs are not necessarily well-structured transition systems, [Kaiser et al., 2014] shows that it is always possible to convert a dual reference program obtained from an asynchronous program via predicate abstraction to a well-structured transition system. An advantage of our technique is that it is complete for refutation, and that we are able to use standard techniques from sequential verification to compute refinement predicates, whereas it is less clear how to automatically generate their inter-thread predicates that are of the form $\varphi(l, l_P)$, where l is the local variable of a distinct active thread, and l_p stand for a local variable of *all* passive threads (and therefore the predicates are universally quantified).

Model checking modulo theories (MCMT) is a general method for proving safety of array-based systems [Ghilardi et al., 2008], which generalize finite-state parameterized systems. The MCMT algorithm is a backwards analysis that uses ideas from well-structured transition systems to guarantee termination for a subclass of array-based systems (namely those that can be expressed in well-quasi ordered theories); this is similar in spirit to the proof checking algorithm presented in Section 5.3 and the covering relation defined in [Ghilardi et al., 2008] is strikingly similar to Definition 5.3.8. In [Alberti et al., 2012], MCMT is combined with interpolation-based abstraction, which in some practical cases terminates when the MCMT algorithm does not. Perhaps the most notable difference between the our approach and MCMT is that we separate the verification problem into two sub-problems: constructing a proof space, and

checking the adequacy of the proof space.

Unbounded Parallelism and Bounded Memory There has been a great deal of work in the area of automated verification of concurrent programs where the number of threads is unbounded but the threads are finite-state [Bouajjani et al., 2000, Pnueli et al., 2001, Namjoshi, 2007, Kaiser et al., 2010, La Torre et al., 2010]. In the setting of sequential programs, model checking algorithms for finite-state systems can be brought to bear on infinite state systems by using predicate abstraction to extract a finite-state model. In the setting of unbounded concurrency, predicate abstraction is not as straight forward: applying predicate abstraction to an asynchronous program with infinite-state threads generally does *not* yield an asynchronous program with finite-state threads. For example, consider a predicate `l < g`, where `l` is a local variable of some thread and `g` is a global variable. The truth of this predicate can be changed by *any* thread (not just the one to which `l` belongs) by assigning to the global variable `g`. Since its truth can be changed by any thread, the predicate `l < g` corresponds to a global variable. If there are infinitely many threads (and thus infinitely many copies of the local variable `l`), this means that there are infinitely many global variables, which is beyond the classical models of multi-threaded Boolean programs. This phenomenon is discussed in more detail in [Kaiser et al., 2014].

In [Abdulla et al., 2010], a border case between unbounded and bounded data is investigated where shared variables range over unbounded domain of naturals but the local states of processes are finite (i.e. local variables of threads range over finite domains).

Automata on infinite alphabets One of the main contributions of this chapter is a method for determining whether the language of error traces of some program P (violating some non-reachability property) are included inside the language of correct traces proved infeasible by some regular proof space $\mathscr{H}$. Classical automata-theoretic techniques cannot directly be applied because the alphabet of program commands is infinite. However, the automata theory community has developed generalizations of automata to infinite alphabets; the most relevant to our work is *(alternating) register automata (ARA)*, which are closely related to predicate automata. Register automata were first introduced in [Kaminski and Francez, 1994]. Universality for register automata was shown to be undecidable in [Neven et al., 2004], which implies ARA emptiness is undecidable in the general case. However, the emptiness problem for ARA with 1 register (cf. monadic predicate automata) was proved to be decidable in [Demri and Lazić, 2009] by reduction to reachability for lossy counter machines, and a direct proof based on well-structured transition systems was later presented in [Figueira, 2012].

Proofs that count Counting proofs are another method for proving safety of programs with unboundedly many threads that is based on the trace abstraction paradigm [Farzan et al., 2014]. Counting proofs automatically synthesize auxiliary variables in complex counting arguments for parameterized protocols. These auxiliary counters are not expressible in proof spaces. On the other hand, proof spaces are capable of proving properties of programs that involve reasoning about infinite-domain local variables which is beyond abilities of counting proofs. It would be interesting to investigate whether the strengths of the counting proofs and proof spaces can be combined into a single framework.

Part IV

Epilogue

Chapter 6

Conclusion

6.1 Summary

This dissertation supports the thesis that tractable and precise algorithmic verification techniques can be built on a foundation that explicitly represents parallelism. We address the explosion in the number of behaviours caused by scheduler non-determinism in the interleaving model of concurrency. Our research advances the state-of-the-art in both static analysis and software model checking by demonstrating how the *parallel proofs for parallel programs* approach can be applied to each discipline.

In Chapter 3, we show how data flow graphs can be used for static analysis of programs with an unbounded number of threads. Data flow graphs are an alternative to the classical control flow graph representation of programs that dominates static analysis. While control flow graphs represent the evolution of a program's control state from one instant to the next, data flow graphs graphs represent causally ordered flow of information. Using data flow graphs for invariant generation separates the problem into two: generating a data flow model of the program in question, and generating invariants from the data flow model. Chapter 3 develops a novel algorithm for solving the former problem and shows how existing techniques can be adapted to solve the latter. The approach has been implemented in the static analyzer DUET and used to prove properties of Linux device drivers. Tractability and precision are demonstrated experimentally: DUET is shown to be able to verify array bounds and integer overflow properties of Linux device drivers and to out-perform existing techniques for verification of Boolean programs with infinitely many threads.

The static analysis presented in Chapter 3 is terminating, but may suffer from false alarms. In Chapter 4, we bring data flow graphs to bear on the problem of software model checking. We present *inductive data flow graphs* (iDFGs), a proof system for concurrent programs with finitely many threads. Where data flow graphs are a truly concurrent model of *programs*, inductive data flow graphs are a truly concurrent model of *proofs*. We show how inductive data flow graphs can be used for software model checking following the trace abstraction paradigm. We demonstrate tractability and precision of the approach analytically, with a theorem (Theorem 4.4.6) demonstrating that iDFGs are a relatively complete proof system and moreover that iDFG proofs are succinct.

Chapter 5 describes *proof spaces*, a system for proving non-reachability properties of concurrent programs with an arbitrary number of threads. Proof spaces can be seen as a generalization of inductive data flow graphs from a finite to an arbitrary set of threads. Data flow graphs (and inductive data flow

graphs) are graphical models that represent parallelism with bifurcating edges. Proof spaces represent parallelism more abstractly, using the Conjunction rule of Hoare logic. Inductive data flow graphs are built on the foundation of classical automata theory, which breaks down in the setting of unbounded concurrency (which requires *infinite-state* automata accepting languages over an *infinite* alphabet). We define *predicate automata*, an infinite-state, infinite-alphabet generalization of alternating finite automata (interesting in their own right), and show how proof checking can be reduced to predicate automata emptiness. We show that predicate automata emptiness is undecidable in general but is decidable for interesting sub-classes of predicate automata that correspond to "weakly-coupled" proofs. As with iDFGs, tractability and precision of proof spaces is proved analytically. The succinctness results of inductive data flow graphs carry over to proof spaces, and Theorem 5.4.2 shows that proof spaces are complete relative to inductive invariants with thread quantification and control predicates.

The interleaving model of concurrency simplifies the theory of concurrent systems, but also makes automated reasoning intractable. The work in this dissertation has been in showing how *true concurrency* can be supported in proof systems for automated reasoning about concurrency systems.

6.2 Retrospective

There are three distinct but related techniques described in this dissertation: data flow graphs, inductive data flow graphs, and proof spaces. All three approach the same problem of *how to effectively reason about multi-threaded software?* Naturally one might ask about the relative strengths of each.

Data flow graphs are capable of capturing information about simple synchronization patterns such as locking. A strength of the data flow graph approach is that this capability arises naturally from the fact that synchronization prevents data flows, rather than specialized heuristics for reasoning about locks. On the other hand, specialized heuristics for known synchronization primitives are likely to be effective in practice. A limitation of the static analysis technique presented in Part II is that it is poorly suited to reasoning about intricate interaction between threads. In particular, since the analysis is limited to generating thread state invariants, it cannot be used to infer relationships between local variables of different threads (such as the property that threads are assigned non-overlapping blocks of tasks in the thread pooling example in Figure 5.1a).

Inductive data flow graphs and proof spaces, on the other hand, are highly capable of reasoning about complex thread interactions. This perhaps the greatest strength of the *parallel proofs* approach over thread-modular techniques, which dominate the literature on formal methods for concurrency. Inductive data flow graphs and proof spaces can exploit thread modularity when it exists (Section 4.4) but are not burdened by the *requirement* of encoding non-modular arguments into modular proofs. The capacity for reasoning about complex thread interactions comes at the price of scalability. The static analysis presented in Part II runs in polynomial time, while the software model checking algorithms in Part III are not guaranteed to terminate. Moreover, Part III makes use of computationally complex sub-procedures, such as emptiness checking for alternating finite automata and Craig interpolation.

A reasonable rule of thumb is that data flow graphs are best for proving simple properties of large code bases, and inductive data flow graphs and proof spaces are best for proving complex properties of small programs.

Chapter 7

Future work

My research addresses some of the challenges of reasoning about concurrent software, and it also suggests a number of directions for future work. This section catalogues a few of these.

Relaxed-memory concurrency The focus of this dissertation is on alleviating the combinatorial explosion in program behaviours incurred by the interleaving model of concurrency. However, identifying behaviours of concurrent systems with interleaved executions of threads is not just inefficient, in some cases it is *wrong*. Programs are typically executed on machines with per-processor caches that do not maintain global consistency. As a result, concurrent systems exhibit phenomena that cannot be attributed to any interleaved execution. A classical example is the following program:

global `x, y`
local `r1, r2`
initially $\texttt{x} = 0 \wedge \texttt{y} = 0$

Thread 1:	Thread 2:
`x := 1`	`y := 1`
`r1 := y`	`r2 := x`

Figure 7.1: A program intended for execution on a relaxed memory model.

Any interleaved execution of this program ends in a state where `r1` is `1` or `r2` is `1` (or both). However, it is possible for both `r1` and `r2` to be 0 after executing this program on most contemporary multiprocessors, since `r1 := y` and `r2 := x` may read stale values of `y` and `x` from their caches.

The static analysis and software model checking techniques presented in this dissertation assume sequential consistency, which is unsound for verification if the program is intended to be executed on relaxed-memory hardware. However, there is potential to extend these techniques to relaxed memory models. The static analysis presented in Chapter 3 is particularly promising in this regard, because data flow graphs have a natural capacity to represent sequentially inconsistent behaviour. For example, a data flow graph for the program above is pictured in Figure 7.2: note that ℓ_1 has two incoming edges labelled with `y`; one from the assignment `y := 1` and one from the initial state. DUET, the tool that implements the invariant generation technique in Chapter 3 already has limited support for weak memory models (enabled by the `-weak` command line flag). However, this support extends only to non-relational analyses on data flow graphs. An interesting direction for future research is to develop techniques for generating

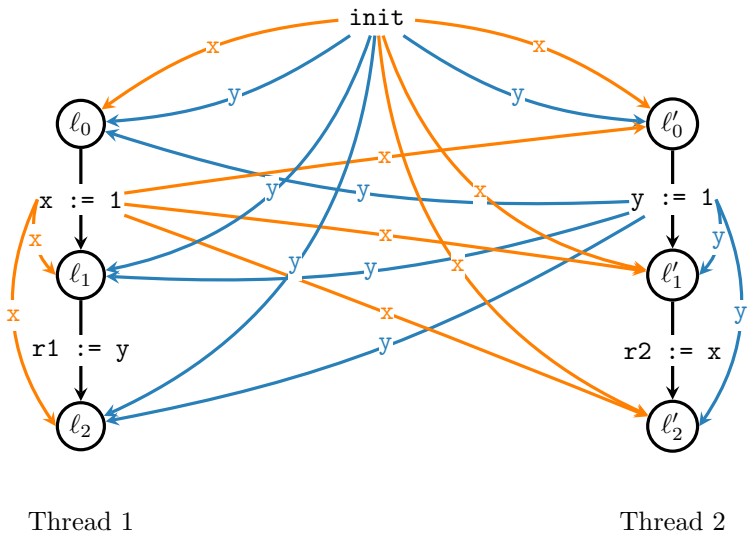


Figure 7.2: A DFG representation of the program in Figure 7.1.

relational data flow graphs under weak memory models.

Efficient implementation of proof spaces Part III gives simple proof-of-concept algorithms for automating inductive data flow graphs and proof spaces. However, these algorithms are not implemented – their effectiveness is proved analytically (Theorem 4.4.6) rather than experimentally. There are two interesting problems that remain to be solved for these techniques to be practical: *construction of proof spaces from traces*, and *emptiness checking for predicate automata*.

The procedure for constructing proof spaces from traces given in Algorithm 6 is heavily dependent on the Interpolate procedure for constructing intermediate assertions. In theory, it is straight-forward to implement Interpolate on top of existing procedures for Craig interpolation [Henzinger et al., 2004b, McMillan, 2005, Rybalchenko and Sofronie-Stokkermans, 2007, Albarghouthi and McMillan, 2013]. In practice, the design of this procedure is crucial to the success of proof generalization, and specialized procedures that take concurrency into account for the construction of intermediate assertions must be designed. One interesting research direction is to experiment with using Horn clause verifiers to generate proof spaces from traces. In particular, it is straightforward to design a system of Horn clauses whose solution yields a proof space that uses only assertions over the globals and locals of one thread. Such proofs corresponds to monadic proof spaces, guaranteeing the success of our algorithm for predicate automata emptiness (Proposition 5.3.15).

In Section 5.3, we develop a proof-of-concept (semi-)algorithm for solving the emptiness problem for predicate automata. The algorithm is a search procedure that attempts to find an accepting configuration that is reachable from an initial configuration. It uses a covering relation to prune the search space, removing a configuration from the search space when it is found to be covered by another configuration. Checking whether one configuration covers another is efficient (polynomial time) for monadic predicate automata, but the general problem is NP-complete (by reduction from the graph homomorphism problem). An efficient data structure for storing sets of PA configurations that makes covering queries efficient is likely to be crucial to the practical success of the proof space technique. Another interesting direction

for future work is to incorporate partial order reduction [Valmari, 1991, Peled, 1993, Godefroid, 1994] into predicate automata emptiness checking.

Support for richer programming languages While the PLIP programming language already exposes some difficult challenges for static analysis and software memory checking (notably, shared memory concurrency, unbounded threads, and ad-hoc synchronization), it is still very much a toy programming language. There are two particularly notable features that PLIP lacks: (recursive) procedures and dynamically allocated memory.

Duet, the tool implementing the analysis from Chapter 3, inlines procedures and uses a simple flow-insensitive pointer analysis to resolve aliasing queries. It is straight-forward to extend Duet to handle recursive procedures in a context-insensitive manner, but context sensitivity is an interesting challenge. Reasoning about the contents of the heap is a problem for which data flow graphs offer an interesting perspective. One of the difficulties in reasoning about the heap is that typical abstract domains represent sets of states by finite-dimensional geometrical objects, with one dimension for each program variable. In the setting of dynamically allocated memory, the challenge is that the heap is unbounded, so there are potentially infinitely many dimensions. Data flow graphs can help solve this problem by representing only information about the dimensions that are relevant at each program point.

The software model checking techniques presented in Chapter 4 and Chapter 5 have no support for reasoning about procedures or the heap. Towards reasoning about the heap, an interesting direction is to adapt proof spaces to use *separation logic* [O'Hearn et al., 2001, Reynolds, 2002] as a base logic rather than first-order logic. Separation logic is a non-classical logic for reasoning about (un)shared resources on the heap. The natural separation logic analogue of the conjunction rule of Hoare logic is the Frame rule:

$$\textsc{Frame}\quad \frac{\{\varphi\}\ \tau\ \{\varphi'\}}{\{\varphi * \psi\}\ \tau\ \{\varphi' * \psi\}}\ \mathsf{mod}(\tau) \cap \mathsf{fv}(\psi) = \emptyset$$

(where $\mathsf{mod}(\tau)$ is the set of variables modified along τ and $\varphi * \psi$ asserts that the heap can be divided into two disjoint parts, with one part satisfying φ and the other ψ). One might consider a variation of proof spaces in which conjunction is replaced by the frame rule. This raises the challenge of supporting the frame rule in predicate automata, or finding a new class of automaton where the frame rule is naturally encoded. [Albargouthi et al., 2015] gives an algorithm for *spatial interpolation* that may prove useful for constructing proof spaces from traces.

The difficulty of analyzing recursive concurrent programs is well-known [Ramalingam, 2000]. In the context of trace abstraction, this difficulty manifests as the problem of developing a class of automaton that can represent both parallelism and recursive behaviour. One potential way forward that avoids this problem is to first treat function calls as atomic actions to generate specifications for them, and then in a separate step verify that functions satisfy their specifications even under interference with other threads. This points to a natural question of whether proof spaces can be used to verify atomicity specifications.

Liveness This dissertation addresses automated verification and refutation of non-reachability properties of concurrent systems. That is, our techniques are designed to prove that a multi-threaded program program never does something bad. Correct operation of concurrent programs additionally requires that

they satisfy *liveness properties*: the program eventually does something good. The problem of verifying liveness properties of programs with unboundedly many infinite-state threads has not been explored in great depth. *Well-founded proof spaces* are a first step in this direction [Farzan et al., 2016]. Well-founded proof spaces extend proof spaces with ranking functions, enabling them to prove termination as well as safety. The proof checking problem for well-founded proof spaces can be reduced to an inclusion problem between *quantified* predicate automata, an extension of predicate automata with quantifiers over thread identifiers. Well-founded proof spaces are a theoretical foundation for proving liveness, but there is a great deal left to explore towards using them for practical software model checking.

Bibliography

[Abdulla et al., 2010] Abdulla, P. A., Chen, Y.-F., Delzanno, G., Haziza, F., Hong, C.-D., and Rezine, A. (2010). Constrained monotonic abstraction: a CEGAR for parameterized verification. In *International Conference on Concurrency Theory*, pages 86–101.

[Abdulla et al., 1996] Abdulla, P. A., Čerāns, K., Jonsson, B., and Tsay, Y.-K. (1996). General decidability theorems for infinite-state systems. In *Logic in Computer Science*, pages 313–321.

[Albarghouthi and McMillan, 2013] Albarghouthi, A. and McMillan, K. L. (2013). Beautiful interpolants. In *Computer Aided Verification*, pages 313–329.

[Albargouthi et al., 2015] Albargouthi, A., Berdine, J., Cook, B., and Kincaid, Z. (2015). Spatial interpolants. In *European Symposium on Programming*, pages 634–660.

[Alberti et al., 2012] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. (2012). Lazy abstraction with interpolants for arrays. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 46–61.

[Alglave et al., 2011] Alglave, J., Kroening, D., He, N., Ranjan, A., Seghir, N., and Tautschnig, M. (2011). CPROVER project. http://www.cprover.org/.

[Ashcroft and Manna, 1970] Ashcroft, E. and Manna, Z. (1970). Formalization of properties of parallel programs. *Machine Intelligence*, 7:17–41.

[Ashcroft, 1975] Ashcroft, E. A. (1975). Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135.

[Berdine et al., 2008] Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., and Sagiv, S. (2008). Thread quantification for concurrent shape analysis. In *Computer Aided Verification*, pages 399–413.

[Bertrand and Miné, 2009] Bertrand, J. and Miné, A. (2009). Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667.

[Bessey et al., 2010] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75.

[Blanchet et al., 2003] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *Programming Language Design and Implementation*, pages 196–207.

[Bouajjani et al., 2000] Bouajjani, A., Jonsson, B., Nilsson, M., and Touili, T. (2000). Regular model checking. In *Computer Aided Verification*, pages 403–418.

[Brückner et al., 2007] Brückner, I., Dräger, K., Finkbeiner, B., and Wehrheim, H. (2007). Slicing abstractions. In *International Symposium on Fundamentals of Software Engineering*, pages 17–32.

[Brzozowski and Leiss, 1980] Brzozowski, J. and Leiss, E. (1980). On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19 – 35.

[Chandra et al., 1981] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. (1981). Alternation. *Journal of the ACM*, 28(1):114–133.

[Cimatti et al., 2011] Cimatti, A., Narasamdya, I., and Roveri, M. (2011). Boosting lazy abstraction for SystemC with partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 341–356.

[Clarke et al., 2000] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169.

[Clarke et al., 2004] Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176.

[Clarke and Emerson, 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, pages 52–71.

[Clarke et al., 1983] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1983). Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Principles of Programming Languages*, pages 117–126.

[Clarke et al., 2005] Clarke, E. M., Kroening, D., Sharygina, N., and Yorav, K. (2005). SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 570–574.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252.

[Cousot and Cousot, 1984] Cousot, P. and Cousot, R. (1984). Invariance proof methods and analysis techniques for parallel programs. In Biermann, A., Guiho, G., and Kodratoff, Y., editors, *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, New York, United States.

[Craig, 1957] Craig, W. (1957). Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22(03):269–285.

[Demri and Lazić, 2009] Demri, S. and Lazić, R. (2009). LTL with the freeze quantifier and register automata. *Transactions on Computational Logic*, 10(3):16:1–16:30.

[Dickson, 1913] Dickson, L. E. (1913). Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422.

[Dijkstra, 1975] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457.

[Dillig et al., 2012] Dillig, I., Dillig, T., and Aiken, A. (2012). Automated error diagnosis using abductive inference. In *Programming Language Design and Implementation*, pages 181–192.

[Donaldson et al., 2012] Donaldson, A. F., Kaiser, A., Kroening, D., Tautschnig, M., and Wahl, T. (2012). Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 41(1):25–44.

[Donaldson et al., 2011] Donaldson, A. F., Kaiser, A., Kroening, D., and Wahl, T. (2011). Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Computer Aided Verification*, pages 356–371.

[Dräger et al., 2010] Dräger, K., Kupriyanov, A., Finkbeiner, B., and Wehrheim, H. (2010). SLAB: A certifying model checker for infinite-state concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–274.

[Emerson and Sistla, 1996] Emerson, E. A. and Sistla, A. P. (1996). Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131.

[ESA, 2014] ESA (2014). How many stars are there in the universe? http://www.esa.int/Our_Activities/Space_Science/Herschel/How_many_stars_are_there_in_the_Universe. Accessed: 2014-08-25.

[Farzan and Kincaid, 2012] Farzan, A. and Kincaid, Z. (2012). Verification of parameterized concurrent programs by modular reasoning about data and control. In *Principles of Programming Languages*, pages 297–308.

[Farzan et al., 2013] Farzan, A., Kincaid, Z., and Podelski, A. (2013). Inductive data flow graphs. In *Principles of Programming Languages*, pages 129–142.

[Farzan et al., 2014] Farzan, A., Kincaid, Z., and Podelski, A. (2014). Proofs that count. In *Principles of Programming Languages*, pages 151–164.

[Farzan et al., 2015] Farzan, A., Kincaid, Z., and Podelski, A. (2015). Proof spaces for unbounded parallelism. In *Principles of Programming Languages*, pages 407–420.

[Farzan et al., 2016] Farzan, A., Kincaid, Z., and Podelski, A. (2016). Proving liveness of parameterized programs. In *Logic in Computer Science*. To appear.

[Ferrante et al., 1987] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

[Figueira, 2012] Figueira, D. (2012). Alternating register automata on finite words and trees. *Logical Methods in Computer Science*, 8(1).

[Finkel, 1987] Finkel, A. (1987). A generalization of the procedure of karp and miller to well structured transition systems. In *International Colloqium on Automata, Languages, and Programming*, pages 499–508.

[Finkel and Schnoebelen, 2001] Finkel, A. and Schnoebelen, P. (2001). Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(12):63 – 92.

[Flanagan et al., 2002] Flanagan, C., Freund, S. N., and Qadeer, S. (2002). Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, pages 262–277.

[Floyd, 1967] Floyd, R. W. (1967). Assigning meanings to programs. In *Symposium on Applied Mathematics*, pages 19–31, Providence.

[Ghilardi et al., 2008] Ghilardi, S., Nicolini, E., Ranise, S., and Zucchelli, D. (2008). Towards SMT model checking of array-based systems. In *International Joint Conference on Automated Reasoning*, pages 67–82.

[Godefroid, 1994] Godefroid, P. (1994). *Partial-Order Methods for the Verification of Concurrent Systems An Approach to the State-Explosion Problem*. PhD thesis, University of Liege.

[Graf and Saïdi, 1997] Graf, S. and Saïdi, H. (1997). Construction of abstract state graphs with PVS. In *Computer Aided Verification*, pages 72–83.

[Grebenshchikov et al., 2012] Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. (2012). Synthesizing software verifiers from proof rules. In *Programming Language Design and Implementation*, pages 405–416.

[Gupta et al., 2011] Gupta, A., Popeea, C., and Rybalchenko, A. (2011). Predicate abstraction and refinement for verifying multi-threaded programs. In *Principles of Programming Languages*, pages 331–344.

[Heizmann et al., 2009] Heizmann, M., Hoenicke, J., and Podelski, A. (2009). Refinement of trace abstraction. In *Static Analysis Symposium*, pages 69–85.

[Henzinger et al., 2004a] Henzinger, T. A., Jhala, R., and Majumdar, R. (2004a). Race checking by context inference. In *Programming Language Design and Implementation*, pages 1–13.

[Henzinger et al., 2004b] Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. L. (2004b). Abstractions from proofs. In *Principles of Programming Languages*, pages 232–244.

[Henzinger et al., 2003] Henzinger, T. A., Jhala, R., Majumdar, R., and Qadeer, S. (2003). Thread-modular abstraction refinement. In Hunt, W. A. and Somenzi, F., editors, *Computer Aided Verification*, pages 262–274.

[Henzinger et al., 2002] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In *Principles of Programming Languages*, pages 58–70.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

[Hoare, 1978] Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677.

[Hoder and Bjørner, 2012] Hoder, K. and Bjørner, N. (2012). Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing*, pages 157–171.

[Johnson and Pingali, 1993] Johnson, R. and Pingali, K. (1993). Dependence-based program analysis. In *Programming Language Design and Implementation*, pages 78–89.

[Jones, 1981] Jones, C. B. (1981). *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University.

[Kahlon et al., 2009] Kahlon, V., Sankaranarayanan, S., and Gupta, A. (2009). Semantic reduction of thread interleavings in concurrent programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–138.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Information Processing Letters*, 74:471–475.

[Kaiser et al., 2010] Kaiser, A., Kroening, D., and Wahl, T. (2010). Dynamic cutoff detection in parameterized concurrent programs. In *Computer Aided Verification*, pages 645–659.

[Kaiser et al., 2014] Kaiser, A., Kroening, D., and Wahl, T. (2014). Lost in abstraction: Monotonicity in multi-threaded programs. In *International Conference on Concurrency Theory*.

[Kaminski and Francez, 1994] Kaminski, M. and Francez, N. (1994). Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363.

[Kildall, 1973] Kildall, G. A. (1973). A unified approach to global program optimization. In *Principles of Programming Languages*, pages 194–206.

[König, 1927] König, D. (1927). Über eine schlussweise aus dem endlichen ins unendliche. *Acta Scientiarum Mathematicarum*, 3:121–130.

[Kuck et al., 1981] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. (1981). Dependence graphs and compiler optimizations. In *Principles of Programming Languages*, pages 207–218.

[La Torre et al., 2010] La Torre, S., Madhusudan, P., and Parlato, G. (2010). Model-checking parameterized concurrent programs using linear interfaces. In *Computer Aided Verification*, pages 629–644.

[Lahiri and Bryant, 2007] Lahiri, S. K. and Bryant, R. E. (2007). Predicate abstraction with indexed predicates. *Transactions on Computational Logic*, 9(1).

[Mazurkiewicz, 1986] Mazurkiewicz, A. (1986). *Petri nets: applications and relationships to other models of concurrency*, chapter Trace theory, pages 278–324. Springer.

[McMillan, 2005] McMillan, K. L. (2005). An interpolating theorem prover. *Theoretical Computer Science*, 345(1):101–121.

[McMillan, 2006] McMillan, K. L. (2006). Lazy abstraction with interpolants. In *Computer Aided Verification*, pages 123–136.

[Miné, 2006] Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100.

[Miné, 2011] Miné, A. (2011). Static analysis of run-time errors in embedded critical parallel c programs. In *European Symposium on Programming*, pages 398–418.

[Miné, 2014] Miné, A. (2014). Relational thread-modular static value analysis by abstract interpretation. In McMillan, K. L. and Rival, X., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 39–58.

[Minsky, 1961] Minsky, M. L. (1961). Recursive unsolvability of post's problem of" tag" and other topics in theory of turing machines. *Annals of Mathematics*, pages 437–455.

[Namjoshi, 2007] Namjoshi, K. S. (2007). Symmetry and completeness in the analysis of parameterized systems. In *Verification, Model Checking, and Abstract Interpretation*, pages 299–313.

[Necula et al., 2002] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. (2002). CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228.

[Neven et al., 2004] Neven, F., Schwentick, T., and Vianu, V. (2004). Finite state machines for strings over infinite alphabets. *Transactions on Computational Logic*, 5(3):403–435.

[Nielsen et al., 1981] Nielsen, M., Plotkin, G., and Winskel, G. (1981). Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13(1):85 – 108.

[Oh et al., 2012] Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and implementation of sparse global analyses for c-like languages. In *Programming Language Design and Implementation*, pages 229–238.

[O'Hearn et al., 2001] O'Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In *Computer Science Logic*, pages 1–19.

[Owicki and Gries, 1976] Owicki, S. and Gries, D. (1976). Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19:279–285.

[Peled, 1993] Peled, D. (1993). All from one, one for all: On model checking using representatives. In *Computer Aided Verification*, pages 409–423.

[Petri, 1962] Petri, C. A. (1962). *Kommunikation mit Automaten*. PhD thesis, University of Bonn.

[Pnueli et al., 2001] Pnueli, A., Ruah, S., and Zuck, L. D. (2001). Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97.

[Ramalingam, 2000] Ramalingam, G. (2000). Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430.

[Ramsey, 1930] Ramsey, F. (1930). On a problem of formal logic. *Proceedings of the London Mathematical Society*, 2(1):264–286.

[Reynolds, 2002] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74.

[Rice, 1953] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, pages 358–366.

[Rybalchenko and Sofronie-Stokkermans, 2007] Rybalchenko, A. and Sofronie-Stokkermans, V. (2007). Constraint solving for interpolation. In *Verification, Model Checking, and Abstract Interpretation*, pages 346–362.

[Sanchez et al., 2012] Sanchez, A., Sankaranarayanan, S., Sánchez, C., and Chang, B.-Y. E. (2012). Invariant generation for parametrized systems using self-reflection. In *Static Analysis Symposium*, pages 146–163.

[Segalov et al., 2009] Segalov, M., Lev-Ami, T., Manevich, R., Ganesan, R., and Sagiv, M. (2009). Abstract transformers for thread correlation analysis. In *Asian Symposium on Programming Languages and Systems*, pages 30–46.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

[Turing, 1936] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58:345–363.

[Valmari, 1991] Valmari, A. (1991). Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, pages 491–515.

[Wachter et al., 2013] Wachter, B., Kroening, D., and Ouaknine, J. (2013). Verifying multi-threaded software with impact. In *International Conference on Formal Methods in Computer-Aided Design*, pages 210–217.

[Weise et al., 1994] Weise, D., Crew, R. F., Ernst, M., and Steensgaard, B. (1994). Value dependence graphs: representation without taxation. In *Principles of Programming Languages*, pages 297–310.

[Whaley and Lam, 2004] Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation*, pages 131–144.

[Witkowski et al., 2007] Witkowski, T., Blanc, N., Kroening, D., and Weissenbacher, G. (2007). Model checking concurrent linux device drivers. In *International Conference on Automated Software Engineering*, pages 501–504.

Index